New Design Companions
Opening up the process through Self-Made Computation

by

Laia Mogas-Soldevila

Architect, Polytechnic University of Catalonia, 2009

Submitted to the Department of Architecture
in partial fulfillment of the requirements for the degree of

Master of Science in Architecture Studies

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author ....
Department of Architecture
May 23, 2013

Certified by ....
George Stiny
PhD, Professor of Design and Computation
Thesis Supervisor

Accepted by ....
Takehiko Nagakura
Associate Professor of Design and Computation
Chairman, Committee for Graduate Students

New Design Companions
Opening up the process through Self-Made Computation

by

Laia Mogas-Soldevila

Submitted to the Department of Architecture
on May 23, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Architecture Studies

Abstract

This thesis is about man and machine roles in the early conception of designs where it investigates computational methods that support creativity and surprise. It discusses the relationship between human and digital medium in the enterprise of Computer-Aided Design[1], and Self-Made Computation to empower the designer as driver of digital processes taking the computer as an active collaborator, or a sharp apprentice, rather than a master.
In a design process tool personalization enables precise feedback between human and medium. In the field of architecture, every project is unique, and there are as many design workflows as designers. However current off-the-shelf design software has an inflexible built-in structure targeting general problem-solving that can interfere with non-standard design needs. Today, those with programming agility look for customized processes that assist early problem-finding instead of converging solutions. Contributing to alleviate software frustrations, smaller tailor-made applications prove to be precisely tailored, viable and enriching companions in certain moments of the project development.
Previous work on the impact of standardized software for design has focused on the figure of the designer as a tool-user, this thesis addresses the question from the vision of the designer as a tool-maker. It investigates how self-made software can become a design companion for computational thinking – observed here as a new mindset that shifts design workflows, rather than a technique. The research compares and diagrams designer-toolmaker work where self-made applets where produced, as well as the structures in the work of rule-maker artisans.
The main contributions are a comparative study of three models of computer-aided design, their history and technical review, their influence in design workflows and a graphical framework to better compare them. Critical analysis reveals a common structure to tailor a creative and explorative design workflow. Its advantages and limitations are exposed to guide designers into alternative computational methods for design processes.

Keywords: design workflow; computation; applets; self-made tools; diagrams; design process; feedback; computers; computer-assisted-design;

Thesis Supervisor: George Stiny
Title: PhD, Professor of Design and Computation

[1]Computer-aided design, also known as computer-aided drafting or computer-aided design and drafting, is the use of computer systems to assist in the modification, analysis, or optimization of a design. Computer-aided drafting describes the process of creating a technical drawing with the use of computer software. CAD software uses vector based graphics to depict the objects of traditional drafting.

Acknowledgments

This thesis is dedicated to my life and work partner, Jorge Duro-Royo, for his constant, generous and wise support, and his challenging motivation. Many thanks to the MIT Design and Computation students for always understanding and encouraging. Thanks to my advisor and readers for helping me think through this complex and rapidly evolving subject.

Laia Mogas-Soldevila has received financial support through the la Caixa Fellowship Grant for Post-Graduate Studies (Caixa d'Estalvis i Pensions de Barcelona la Caixa, Spain).

This master thesis has been examined by a Committee of the Department of Architecture as follows:

Professor Takehiko Nagakura . . . .

Chair and Reader, Thesis Committee
Associate Professor of Design and Computation

Professor George Stiny . . . .

Thesis Supervisor
Professor of Design and Computation

Professor Mark Goulthorpe . . . .

Reader, Thesis Committee
Associate Professor of Design

Professor Mine Ozkar . . . .

Reader, Thesis Committee
Visiting Associate Professor of Design and Computation

Contents

Chapter 1

Introduction

One programs not because one understands, but in order to come to understand. Programming is an act of design. To write a program is to legislate the laws for a world one first has to create in imagination. – Weizenbaum, 1976.

In architecture there is no mass manufacturing of projects, every single one is different. This is why design is "a research-related and knowledge-producing process". Today, engineering fields – such as structuring, digital tectonics, digital morphogenesis, materiality or performance-driven evolutionary generation – are also common to the architect. In consequence, during the last decade, interdisciplinary research groups like the "Arup Advanced Geometry Unit (AGU) deal with the new range of geometric, computational and materialization problems of contemporary design practice" [Oxman, 2010]. One of the purposes of these 'in-house' computational research groups is to come up with personalized software tools that fit the needs and assist the know-how of a particular architecture office, or even a particular project.

A good example of a tool built for a specific project is the structural design research by Chris Williams[1] on the British Museum Great Court renewal. Williams writes; "the spiraling geometry of the steel members was generated working closely with the architects, Foster and Partners, and the engineers, Buro Happold. A combination of analytic and numerical methods were developed to satisfy architectural, structural and glazing constraints. Over 3000 lines of computer code were specially written for the project, mainly for the geometry definition, but also for structural analysis" [Williams, 2001]. The conflicting constraints pointed out by Williams, are one of the reasons where self-made computation can better assist the task of exploring a design space of solutions. With less resources than Ove Arup or Foster and Partners, every professional business, design school or academic department also require different software tools in terms of size, functionality, interface, field, etc., to tackle their design research needs. Today, there is an emerging generation of designers that build tools to better understand and explore with the computer.

There is an upcoming shift in the way we look at the computer role and the human role in design computation. This document leads the reader through a discussion about the emergence of a designer tool-maker – instead of a designer tool-user –, a professional who sees the computer as a companion, who builds rule-based

[1]Dr Chris J K Williams, MA, PhD, is a structural engineer who worked for Ove Arup and Partners prior to joining the Department of Architecture and Civil Engineering at the University of Bath, UK (http://www.bath.ac.uk/ace/people/williams/index.html).

computational systems to dialogue with it, who embeds Applets[2] in the design process[3] to reveal a moment of discovery – a moment that redefines the way we look at both the work and at our imagination. In summary, this thesis investigates the value of embedding self-made software into the design process.

1.1 Context and Need

> *We cannot shape what we don't understand, and what we don't understand and use ends up shaping us. – Catarina Mota, 2012.*

Catarina Mota[4], open-source advocate, maker and research scholar, believes that we should have a better understanding of the components that make the world around us. One of her aims is to facilitate open source technology transfer to help create resilient communities around the world. She addresses the benefits of opening scientific research, on new materials and technologies, to non-expert communities of makers (Figure 1-1). Makers "create out of passion and curiosity, are not afraid to fail, and often tackle problems in unconventional ways and in the process end up discovering alternatives or even better ways to do things" [Mota, 2011], which is exactly what happens when designers create their little design tools to invent with the computer. The effort in acquiring the knowledge to create these self-made software tools, and the process of making them, will provide a mindset shift from tool-user to tool-maker.

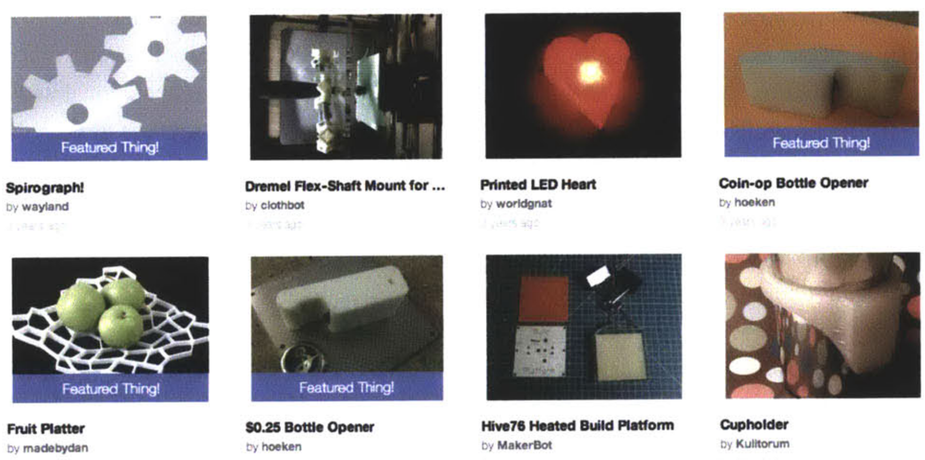


Figure 1-1: One of the user sites where to learn how non-experts make smart materials or custom electronics to better understand the components that make the world around us [Mota, 2012]

[2] An applet is a software application that performs only a small set of tasks. Or also a small application program that performs a simple function within a larger body of software such as a web browser or operating system. [Oxford English Dictionary, 2008].

[3] "The design process is presumed not to be a prescriptive situation but something unique to the designer. The semantic difficulty comes from the use of the word process, process implies something that can be defined, and automated" [Burry et al., 2000], and design is cannot be. Examples of the understanding of design processes are provided in Chapter 2.

[4] Portuguese open source advocate, maker, and research scholar. Catarina co-founded Open Materials (do-it-yourself smart materials), Everywhere Tech (open source technology transfer), and AltLab (Lisbon's hacker space). She has taught numerous hands-on workshops on hi-tech materials and simple circuitry with the goal of encouraging people with little to no science background to take a proactive interest in science, technology and knowledge-sharing. Previously, she co-chaired the Open Hardware Summit 2012, served on the board of directors of the Open Source Hardware Association, taught as an adjunct faculty member at ITP-NYU, and was a fellow of the National Science and Technology Foundation of Portugal. Catarina is wrapping up her PhD dissertation on the social impact of open and collaborative practices for the development of physical goods and technologies. She is currently a visiting scholar at ITP-NYU, Research Chair at the Open Source Hardware Association, TED Fellow, and member of NYC Resistor.

1.1.1 Emerging Cultures

In recent years we have witnessed the emergence of the "maker culture", the first stages of manufacturing democratization that promises to revolutionize the means of design, production and distribution of material goods and give rise to a new class of creators and producers. "A disruptive technology and several cultural and economic driving forces are leading to what has already been called a new industrial revolution: public access to digital fabrication tools, software and databases of blueprints; a tech Do-It-Yourself movement; and a growing desire amongst individuals to shape and personalize the material goods they consume" [Mota, 2011]. Neil Gershenfeld[5] describes a close future in which everyone will have a personal fabricator, a machine capable of producing not only material objects but also other machines that make things [Gershenfeld, 2005]. Can we also be witnessing a "tool-maker culture"? In the same way makers make machines that make things, can designers make the tools that help them think about design?

There is today a paradigm shift in the world of software tools that is changing the human role from driven to driver, from tool-user to tool-maker in computer-assisted design. During the last ten years, with the intention to render sophisticated software more intuitive and usable by those who do not consider computers natural allies, the interfaces between users and software have become more and more friendly. As a result, the designer is much less exposed to the internal workings of the computer programs. Software developers try to assist the needs of a broad range of CAD users, making their interfaces and functionalities as general as possible, which causes that designers wishing to experiment with the computer are not likely to find assistance in these tools. In response, there has been for a while an emerging "scripting culture", described by Mark Burry[6] in a recent book with the same name[7], where he argues that "initiation in scripting for CAD has already started in education and practice, questions like "this could well be too difficult and beyond me", or "how would I use it anyway?" started being answered, and scripting or visual programing for CAD is now becoming mainstream" [Burry, 2011]. This is reflected in the online communities of "scripters" that support that programming – or more precisely customizing software through scripting – "provides a range of possibilities for creative speculation that is simply not possible using the software only as the manufacturers intended it to be used. Because scripting is effectively a computing program overlay, the tool user (designer) becomes the new toolmaker (software engineer)" [Burry, 2011].

In order to avoid being inadvertently restricted by a software apparent flexibility, there is today another culture emerging, the "applet-making culture". Once a designer has the computational thinking ability to program within CAD and automate and experiment with the software package capabilities, he is in a better position to work creatively in tandem with the computer, the gap between user and software engineer is narrowed. But applet-makers feel the need to go one step further because their design problem is different from the ones that can be approached by generic solutions. It is not yet defined, they need to embark in a problem-finding adventure. The rules that describe applets tend to have conflicting constraints and an intuition of shape that escapes human imagination. In order to explore this broad design space, they write software applications that perform only small sets of tasks, that stand alone, out of CAD packages, and demand a flexible structure not covered

[5]Neil Gershenfeld is a professor at MIT and the head of MIT's Center for Bits and Atoms, a sister lab spun out of the popular MIT Media Lab.

[6]Mark Burry is a New Zealand architect, Professor of Innovation and Director of the Spatial Information Architecture Laboratory and founding Director of the Design Research Institute at RMIT University, Melbourne, Australia. He is also executive architect and researcher at the Temple Sagrada Famlia in Barcelona, Catalonia, Spain. He has received degrees from the University of Cambridge.

[7]"Scripting Cultures" by Mark Burry in 2011, see Bibliography for complete reference.

by scripting. Small applet devices liberate the exploration from the constraints that generic problem-solving packages may impose to certain non-standard design approaches. The characteristics of these applet-making cultures, and the frustrations they respond to, will be addressed in Chapter 3.

With the same do-it-yourself experimentation philosophy that Mota proposes with smart materials, tailored computation can enable a new inventive process, a process that involves the computer as a digital design agent with a contribution to the design process, surpassing representation in an emergent collaboration between designer and medium. It envisions that designers can participate with the computer in a bidirectional[8] design model [Kilian, 2006], and that its feedback incorporates new ways to look at designs.

But, what do I mean by feedback – or bidirectional process – in a man-machine relationship?

Ashby claimed in 1956 that "feedback exists between two parts when each affects the other" (Figure 1-3). Feedback in a man-machine relationship for design comes from reversing the roles of the computer-as-brain and designer-as-apprentice. Could we let the machine be our apprentice? Can the computer make proposals that the designer craftsman can critique, with the expectation that the craft will be improved? Referring here for example, to the powerful stimulus that provocative student questions in the design studio can offer to the experienced professor.

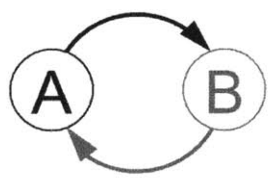


Figure 1-2: Feedback exists between two parts when each affects the other (W. R. Ashby, "An introduction to cybernetics", 1956)

1.1.2 'D' for Design

As we will see in Chapter 2, current Computer-Assisted Design (CAD) tools are very well suited for representation and evaluation purposes but of not much help as conceptual or feedback-providing companions for the designer [Sola-Morales, 2000]. The 'D' in the acronym CAD would be more accurately defined as drafting, not design (representation, not synthesis) [Burry et al., 2000]. For now lets assume that CAD operates on geometry, it assists the designer in the process of sketching and refining ideas in the world of lines, circles, surfaces, meshes, fillets, boolean operations, solids, etc. Paradoxically the same CAD tool is used while exploring designs, and also while automating drafting. Inevitably, because of the very nature of these softwares, the exploration intents will be constrained by limited functionalities – coming from the geometric kernel, a concept further discussed in Chapter 2 –, for generic drafting, that some expert has prepared for us to use.

[8]Bidirectional Models where explored by Axel Kilian in his PhD thesis Submitted to the Department of Architecture on Jan 13, 2006. He explains that today digital models for design exploration are not used to their full potential. The research efforts in the past decades have placed geometric design representations firmly at the center of digital design environments. In his thesis he argues that replacing commonly used analytical, uni-directional models for linking representations, with bidirectional ones, further supports design exploration. The key benefit of bidirectional models is the ability to swap the role of driver and driven in the exploration.

But is design only Geometric Drafting?

What if the designer is interested on investigating light, or color, or image analysis, or weight, or comfort, etc. How do these aspects translate to CAD? Their interplay may be ultimately represented by geometry, but not necessarily explored through it. Back to Weizenbaum's quote at the beginning of this Chapter, "one programs not because one understands, but in order to come to understand. Programming is an act of design. To write a program is to legislate the laws for a world one first has to create in imagination" [Weizenbaum, 1976]. Maybe we are not even capable of imagining that world, but we can describe some rules, intuitions and conflicting constraints to explore it. Can we invent with the computer through rules and a language other than CAD's?

In the context of this thesis, rules are what designers describe in order to explore with the computer. With the richness that "a limited number of rules can generate an unlimited number of different things" [Stiny, 2006]. I argue that designing is not combining, even if we use programming to build our generative tools. I approach it here as a research-related and knowledge-producing process, and I argue that an early exploration can be accompanied by the computer.

How do I know what I am going to use until I start? Where's the creativity? Can I apply rules to calculate without combining symbols? – George Stiny, 2006

George Stiny [9] claims that one issue with the majority of existing information processing models of design and computation is that they have a unique description, in terms of standard parts, that is fixed before the computation begins and kept constant as it proceeds. Another issue is the fact that these parts are independent in combination, being the only ones that are recognized and used as rules are applied. In Shape Grammars[10] "the parts of a shape and their consequent possibilities for organization are indefinite, being contingent on rules and the transformations used to apply them" [Stiny, 1994]. Stiny's main interest has been to figure out how to calculate without units or symbols, and his solution "to see how to calculate with shapes" [Stiny, 2006].

As I focus on informed design, and on explorations involving conflicting constraints, the learning process while rule-making, and the possibility to go back to the coded rules and change them anytime, is what enables the designer "to see" differently.

To get there though, there must be some symbolic communication with the machine. The communication itself is writing code, that when executed, initiates a conversation. The designer sets a problem description (describes, not prescribes) like in shape grammars – and allows the computer to explore the design space in response. Then the human "sees" the computer proposals and changes the rules or makes up new ones, discovering other ways of looking at the problem. Then answers back in a feedback process of communication between man and machine, between rule-maker and companion.

[9] George Stiny is a design and computation theorist, MIT Professor of Design and Computation and advisor for the present thesis.

[10] In his book "Shape: Talking about Seeing and Doing", Stiny explains why shapes are so ambiguous and why their lack of structure is so crucial for design. Stiny shows that grammatical rules for shapes can produce an infinite complexity of designs.

1.1.3 Notes on Applet-Making

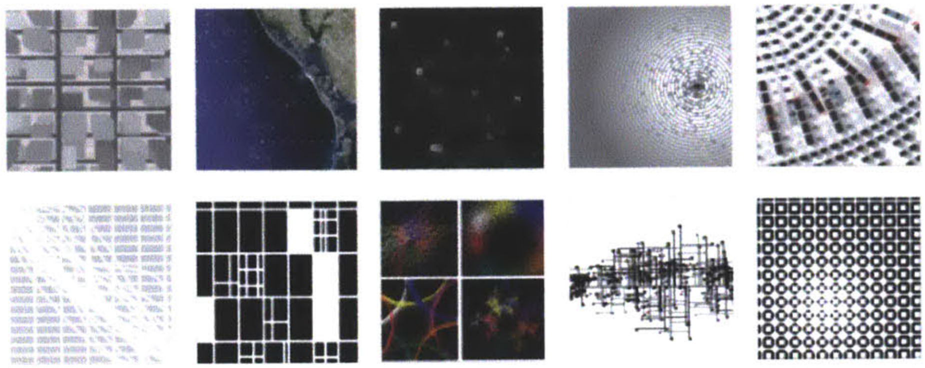


Figure 1-3: "Beyond Measurement", Urban patterns exploration through Procesing applets, challenging the idea of urban and architectural "modeling" between simplified abstraction and cinematic simulation. Students learn programming skills to transform and choreograph their own urban settings through studies and constructions of projecting (instead of descriptive) digital modeling.(Department of Architecture, Cornell AAP, Accessed through the OpenProcessing site, 2011)

Maybe, one of the clearer initiatives that explore design with the computer beyond CAD, is that of the Processing[11] framework by Ben Fry and Casey Reas. When exploring the possibilities expanded by open source tools and the importance of their growing online communities, the Processing environment is a way of bringing "what is exciting about software into the visual arts at a level where people that are visually motivated and focused can explore, experiment, and play with code". The Processing applet-maker community is described as "not expert technicians but visual thinkers with creative ideas" [Alejos and Muscinesi, 2013]. The platform provides backbone functionalities and allows for the designer to create new ones[12].

Then, how is Rule-Making, or more specifically, Applet-Making done?

Applet-Making refers to coding self-made tools to design with. It agrees with Paul Pangaro's[13] "design the designing" model and with the Processing initiative – as we will se in Chapter 2 and 3 – in the sense that more emphasis is given to the construction of open-ended tools that favor early explorations, rather than premature representation of design ideas.

New ways of participating with the computer refer to writing design rules in the form of code to investigate and invent. It is important to note the difference between adapting to existing software constraints and making software for our own constraints. In 1987, Bill Mitchell[14], co-authors a book called "The Art of Computer Graphics Programming" with the subtitle; "A Structured Introduction for Architects and Designers". The book is addressed to those who want to understand computer graphics, "graphics and design professionals: architects, urban designers, landscape architects, interior designers, product designers, graphic artist,

[11]Processing is an open source programming language and environment for people who want to create images, animations, and interactions. Initially developed to serve as a software sketchbook and to teach fundamentals of computer programming within a visual context, Processing also has evolved into a tool for generating finished professional work. Today, there are tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning, prototyping, and production.

[12]In the Processing language,there are two categories of libraries. The core libraries (Video, OpenGL, Serial, Net) are part of the Processing distribution, and contributed libraries are developed, owned, and maintained by members of the Processing community. Occasionally, a contributed library might make its way into the regular distribution.

[13]See Section 2.3 for further detail on Paul Pangaro's research claims on reformulating the traditional Design Thinking paradigm into a Design for a Conversation paradigm.

[14]William J. Mitchell was a Professor of Architecture at Harvard University and a principal of the Computer Aided Design Group (Los Angeles software development firm) at that time.

animators and others" and its "central objective is to teach how to write concise, elegant, well-structured computer programs to generate graphic displays". At that time, when they envisioned that cultural life would be changed profoundly as "medieval culture was shaken to its foundations by the invention of printing and by the mass literacy that followed" [Mitchell et al., 1987], there was not yet a division between tool-makers and tool-users. The designer was responsible for his computer graphics programs, for his computer aids to design.

As Figure 1-4 indicates there is today a tendency to reconnect tool-using – what mostly design disciplines do – and tool-making – what mostly computer experts do, evidencing the need to interface between the two worlds in order to better understand computation for design.
There is an agile generative scripting culture emerging – that operates within CAD tools –, and I claim that designers can go one step further, beyond CAD's complexity and constraints, by self-making small software tools that aid in better understanding their design medium; the computer.

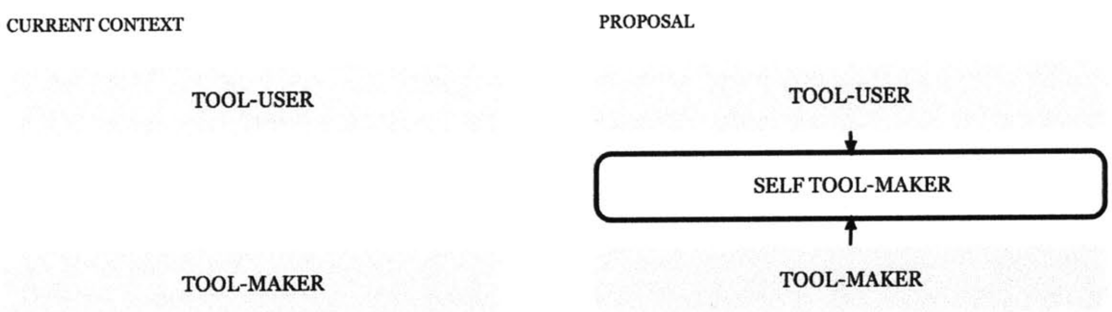


Figure 1-4: Since the development of CAD, the tool-maker and the tool-user are two different figures, but it is today more viable than ever to emphasize the hybridization of the designer taking the best from USING software and MAKING software worlds.

This same philosophy governs the work of Mark Downie or Panagiotis Michalatos[15]. Their methods and motivations, as well as the ones from other contemporary precedents, will be deeply analyzed at the beginning of Chapter 3. Suffice to say here, that they make their own applets – for image processing and structural design – to think with, to reveal aspects and form not enabled by traditional tools.

They are designer-tool-makers that create in the intersection of active (tool-making) and passive (tool-using) computation, whose inventions with the computer are worth exploring. There is an emergent community of designers, like them, building self-made computation to personalize the medium that they use and adapt it to the demands of every different project. They are located in the space of opportunity between tool-using for design and tool-making for software development, taking a mixed approach in the computational abstraction hierarchy by building on the current scripting agility and incorporating the software development logics (Figure 1-5). In Chapter 3, this differentiation will be further detailed.

[15]See Chapter 3 for details on work by Mark Downie or Panagiotis Michalatos.

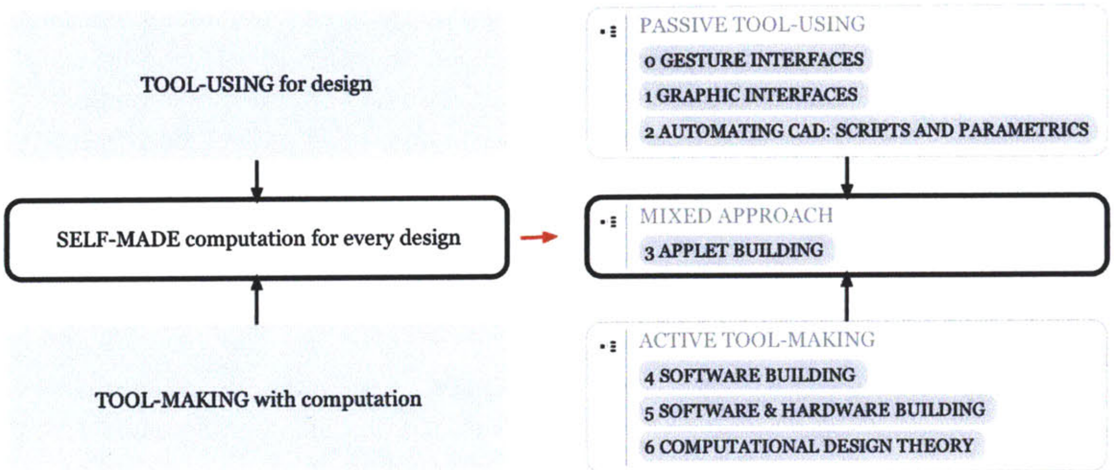


Figure 1-5: Self-Made Computation for Design lays in the intersection of ACTIVE and PASSIVE computation. Applet building characterizes a mixed approach in between Scripting and Software Engineering.

In the same way that Theodor Nelson stated "we must understand computers now" – in 1974, when computers where only used by scientists and before anyone had one at home –, today we must understand design software tools in order to be aware about prescribed constraints that can get in the way of the designer's imagination.

> *We must understand computers now. – Theodor Nelson, 1974*[16].

Instead of merely getting good at using standard software, we can understand its foundations and assumptions, and with more and more proficiency write our small applets to explore other ways of designing with the computer. Because, back to Mota's quote, "what we don't understand and use ends up shaping us", shaping our imagination and designs without us even noticing.

1.2 Intended Contributions

This thesis discusses contemporary questions about the integration of self-made computation in the "creative process". It discusses the subject in two scales of detail; through a general mindset shift emphasis and through a concrete graphical method to depict it.

The mindset shift addressed here spans education practice and scholarship in the architecture discipline. In education it emphasizes teaching computational design thinking to support process-based rather than result-oriented design studios. In architectural practice, even if the professional office calls for shortcuts, it focuses on producing tailored tools, in specific moments of a project development, as the building of a growing repertoire of digital design companion modules that will better assist the practice's design intent. Finally, scholarship research on small self-made rule-based modes of dialogue with the computer can lead the way of

[16]Theodor Holm Nelson (born June 17, 1937) is an American sociologist, philosopher, and pioneer of information technology. In his book from 1974 "Computer Lib", Nelson writes about the need for people to understand computers deeply, more deeply than was generally promoted as computer literacy, which he considers a superficial kind of familiarity with particular hardware and software. His "Down with Cybercrud" is against the centralization of computers such as that performed by IBM at the time, as well as against what he sees as the intentional untruths that "computer people" tell to non-computer people to keep them from understanding computers.

digital design innovation.

The use of a thesis-wide graphical language to unify, analyze and compare design processes is one of the contributions. It is a graphic guide to help designers think about the use of standard design software, scripting or parametric processes, and self-made computation strategies (Figure 1-6). It aims to provide future designer-toolmakers with a support in which to explain their thought processes and compare their computational thinking methods to traditional ones.

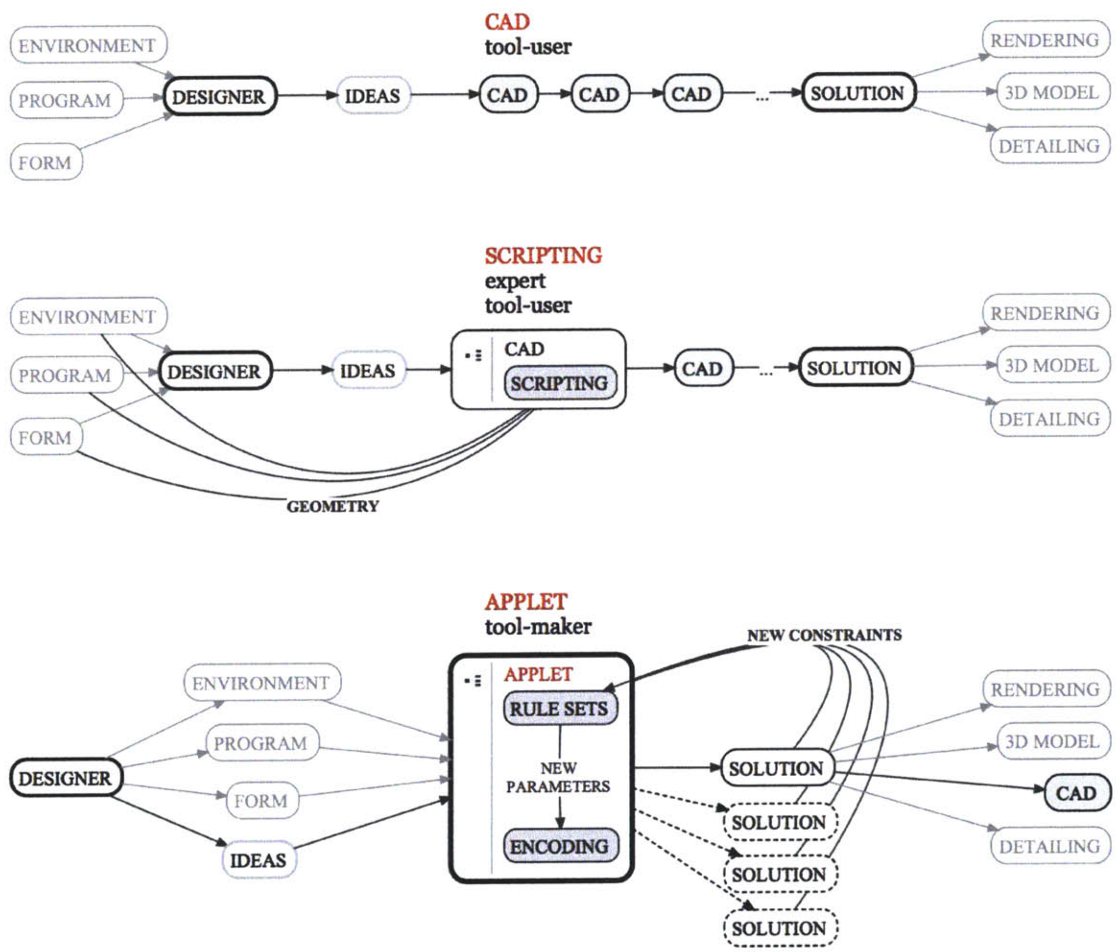


Figure 1-6: CAD is a computerized design aid where the designer is a tool-user of a digital drawing environment. SCRIPTING is a way to access CAD backbone functionality and customize it and automate it where the designer becomes an expert user of the tool. In APPLET-making, the designer is the tool-maker of a small computational device to expire with the computer before CAD representational power comes in the design workflow.

Previous work on the impact of standardized software for design has focused on the figure of the designer as a tool-user, this thesis addresses the question from the vision of the designer as a tool-maker. It investigates how self-made software can become a design companion for computational thinking – observed here as a new mindset that shifts design workflows, rather than a technique. The research compares and diagrams designer-toolmaker work where self-made applets where produced, as well as the structures in the work of rule-maker artisans.

The main contributions are a comparative study of three models of computer-aided design, their history and technical review, their influence in design workflows and a graphical framework to better compare them. Critical analysis reveals a common structure to tailor a creative and explorative design workflow. Its advantages and limitations are exposed to guide designers into alternative computational methods for design processes.

1.3 Structure and Development

The work is organized as follows. Background research and literature closely related to the thesis proposal is provided in Chapter 2. The first section, 2.1, "Behind Computer-Aids to Design", provides context surveying the computer-aided design software intentions in the past. The second section 2.2, "Reconquering Tools", explains the thesis need, answers frequent questions, and discusses current issues and perspectives of modern computational design. And the third one, 2.3 "The 'Design Workflows" wants to evidence a current shift from problem-solving to problem-finding in the scholar representation of design processes.

The thesis core is developed in Chapter 3. The first section 3.1, "Rule-Making Artisans", provides precedents and focuses on other disciplines where rule-making has significantly contributed to the creative process. The section 3.2, "New Design Companions", frames applet-making for design and provides case examples. The last section 3.3, "Scope and Detail", is dedicated to technical details about self-made computation and its scope in digital abstraction.

Finally, Chapter 4 elaborates on the new mindset of the designer-toolmaker, and on the next steps for education, practice and scholarship. It also provides future perspectives on an emerging way of thinking about the design process.

Chapter 2

Background

Our interest is in how an algorithm, loaded with design intent, emerges from the design problem rather than simply the architecture emerging from a known algorithm.
–Roland Snooks[1], 2011

Computation can help us investigate new rules for our designs, we can even "come back later and apply different rules to change them. It becomes very dynamic, a very open-ended kind of process" [Stiny, 2012]. In this thesis the argument is that self-made computation can help us understand what is essential and behind of standard tools and enable a way of thinking beyond representational constraints. It envisions self-made computation as a change of mindset and as a new companion to design with, tailored and disciplined for a specific project but that will inform future ones.

Do new tools imply new tasks? Fifteen years ago, when computer-assisted design software became broadly used in architecture practice it did not tackle any explorative demands but provided drafting automation. "Usually a new tool is used to do things pretty much as they always had been done: usually a new task is done for quite some time by means of adapting existing tools" [McCullough, 1996]. Invention and innovation are gradual, often computers are installed in offices, factories and schools without changing what is actually done in those places. Technology alone will not change the world, limiting our work to whatever shows up in our instruments is inappropriate, we should not abandon the old freedoms when a new technology appears, we should consciously explore the new ones. Even if the majority accepts new technology, only a minority truly adopts new practices.

As Malcolm McCullough[2] claims in 1996, "computer ownership does not guarantee computer literacy" [McCullough, 1996] A tool is for serving intent, it does only what you tell it to do, it is never out of control. The rigidity of some production lines result of the Industrial Revolution has caused use of technology to be equated with unimaginative, semiskilled labor such as attending machines. The same outlooks about indus-

[1]Roland Snooks is a director of the architecture practice Kokkugia. Associate Professor at the University of Pennsylvania, he has directed design studios and seminars at the Pratt Institute, Columbia University GSAPP, SCI-Arc, UCLA, RMIT University, and the Victorian College of the Arts. His current teaching and research interests focus on emergent design processes involving genetic and agent based techniques. He has previously worked in the offices of: Reiser + Umemoto, Kovac Architecture, Minifie Nixon, and Ashton Raggatt McDougall. His work with Kokkugia has been published and exhibited internationally, including at the Beijing Biennale and Chernikhov Prize in Moscow, in addition to being named the Australasian Curators for the 2008 Beijing Biennale.

[2]Malcolm McCullough is a professor at the University of Michigan's Taubman College of Architecture and Urban Planning. He has lectured widely on Urban Computing and place-based Interaction Design.

trial automation have been carried over to computer technology. "The computer has been treated as a mere machine in the mechanical sense of a device that determines its own scope and pace once it is set in motion by a programmer /attendant" [McCullough, 1996], and this mindset regards the whole computer as a single tool.

Computing is not at all monolithic, in software, tool metaphors are usually the main interaction strategy and toolkits the main organizational schema, even data is a tool, so the computer is not a tool, but hundreds. However, task automation in some large business organizations leads the user to perform only one task, use only one tool in its desk computer. For example engineering and architecture firms have been prone to regard CAD as equipment, which then they hire paraprofessionals to "operate", work that consist in putting sketches already completed by hand onto the computer. In 1996, such drafting automation was already perceived as "a natural outgrowth of the leftover industrial-era attitudes about technology" [McCullough, 1996].

2.1 Behind Computer-Aids to Design

Can anyone remember what it was about design that needed aiding before we had Computer Aided Design? With CAD we solved the wrong problem we had a solution but no problem. Now that we have more computing power and extraordinary devices and new interfaces, we still do not know how to define the problem.
–John Frazer "Accelerating Architecture. Day 4: The Computer's Tale", 2005

"Just as log tables supplanted the abacus, the slide rule the log table, the calculator the slide rule, cold clumsy CAD finally exiled the drawing board and instrumental technical drawing told from almost all offices worldwide" [Burry, 2011]. The computer is considered to have entered all disciplines as a provider of workload relief, productivity increase and conceptual renovation. In engineering companies CAD – or Computer-Aided Design software – was introduced in the early 1970's and it responded to a business need to port building information from a computer system to another [Sola-Morales, 2000] [Burry, 2011].

General Motors, commonly known as GM – an American multinational automotive corporation among the world's largest automakers – was an early computer user, using punched card machines as early as 1952 for engineering analysis. They soon were convinced that automation was a solution to problems such as slow hand modifications in car drawings, broad solutions library accessibility, correction of human errors in drafting processes etc. In 1959 the Data Processing department of GM Research started developing a system to store diagrams for rapid retrieval and simple modifications. The idea was that the diagrams would be digitized into the computer, displayed interactively to allow rotations, scaling and projections, and then printed on demand. Lookups would be handled via punched card queries, which would allow operators to quickly retrieve documents for manipulation into whatever local format the user needed, and then print it. Repetitive queries could be automated simply by saving the card stack. IBM was brought in as a partner in 1960 and the research resulted into DAC-1 (Design Augmented by Computer), that was one of the earliest graphical computer aided design systems, released to production in 1963.

McDonnell Douglas (MCAuto), founded in 1960, was a major American aerospace manufacturer and defense contractor, producing a number of famous commercial and military aircraft. McAuto's engineers closely observed the DAC program and the ITEK Electronic Drafting Machine optics design efforts, the latter resulting in the Control Data Digi-graphics commercial CAD system. Their research played a major role in CAD development with the introduction of the sophisticated CADD program that was optimized for the design of aircraft structural components. Because of its special internal needs, the aircraft industry has produced some of the world's leading CAD programs. These include proprietary software developed at Boeing, CADAM by Lockheed, McAuto by McDonnell Douglas and CATIA by Marcel Dassault in France [Carlson, 2007].

Figure 2-1: 1964 Design Automated by Computer (DAC-1), produced by GM and IBM as a response to the automation of certain design tasks in automobile engineering.

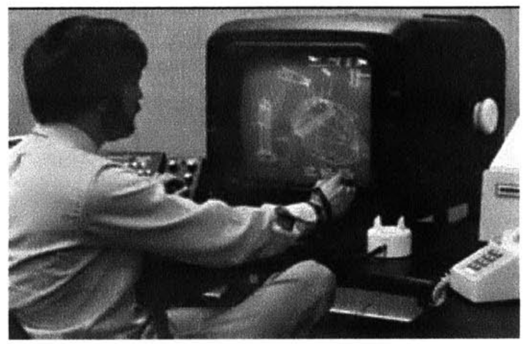


Figure 2-2: 1966 McAuto CADD software, optimized for the design of aircraft structural components, with a major role in CAD development.

Engineering firms where pioneers in the development of computer aided design and its development stayed majorly in the hands of software engineers. In architecture practice CAD was introduced as a helper for creation, modification, analysis or optimization of a design [Menges and Ahlquist, 2012]. However, it became soon clear that CAD's optimization goal was not applicable to the conflicting constraints, qualitative requirements and ever changing intentions that surround an architectural problem. Software packages like BOP[3] maximized the benefits of building business (Figure 2-3), Neil Harper explains about BOP's implementation that "the practical problem of building design can be formulated, in a general way, as an optimization problem. The objective could be minimum initial cost, life-time cost, operating cost, maximum return on in- vestment, etc. This objective is, of course, subject to a variety of constraints due to client program, site limitations, space allocation, design considerations, code constraints, environmental factors, engineering requirements, financing, etc. Hence the practical problem of finding an "optimum" building is in some ways quite similar to the classical problem of mathematical optimization" [Harper, 1968]. A problem formulation, far from explorative design approaches.

[3]BOP (building optimization program) by SOM (Skidmore, Owings and Merrill LLP) one of today's largest architectural firms in the world founded in Chicago in 1936.

Figure 2-3: 1977 Bill Kovacs and Doug Stoker in SOM Computer Room operating BOP, a CAD software package to maximize building business, limiting the solution space to one morphological type and discouraging exploration of any innovative ideas.

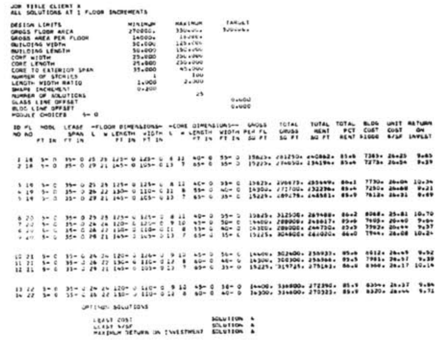


Figure 2-4: BOP's overall summary of Computer Generated Solutions, printed report of the "internal design limits as they currently exist, and the solution geometries which were found to fit within these limits" [Harper, 1968].

Vladimir Badjanak, in 1971, conducted an experiment with students at UC Berkeley and observed that "no desirable causality effect could be attributed to the use of BOP. It in fact limited the solution space to only one morphological type and discouraged exploration of any innovative ideas. Instead of encouraging variety, the underlying philosophy of BOP is to make all high-rises into an average building" [Bazjanac, 1975]. Computer-aided design has been proven to be very suited for documenting completed designs – but of no help and very frustrating during the design phase [Stiny, 1990][Sola-Morales, 2000]. In fact I argue that current design software has a robust built-in structure that may interfere with the explorative needs of the designer and the design process.

2.1.1 The importance of Geometric Kernels

By nature, CAD's structure tries to tackle broad geometric problems, which as many designers as possible can encounter. It is also a result-and-representation-oriented tool rather than explorative and open-ended. In this sense, design takes a back seat to technology; so sometimes "if you just look at the work of many young designers, you can immediately see what sort of CAD system they are using" [Mitchell, 2009]. Why is that? Because of something called geometric kernels[4] that make our applications communicate with the computer memory, processor and external devices (Figure 2-5).

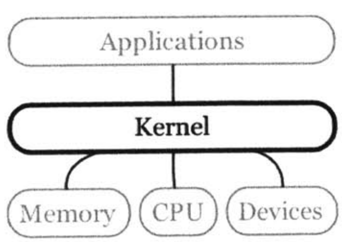


Figure 2-5: The Kernel function is to bridge applications and core processor, memory and external devices in computers.

The geometric kernel in CAD software platforms lists the geometric operations that are made available to

[4]In computing, the kernel is the main component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level. As a basic component of an operating system, a kernel can provide the lowest-level abstraction layer for the resources.

the users of CATIA, GC, Autocad or Rhinoceros through buttons or commands in user interfaces. One of the signs of an expert CAD user is his ability to work around limitations in the kernels of his programs to avoid modeling failures. Also the kernel is key to the ability to compute 3D shapes and models and output 2D drawings from 3D geometry. So the CAD user must be aware of what impacts, a change in this core code, can have on his companys existing and future designs.

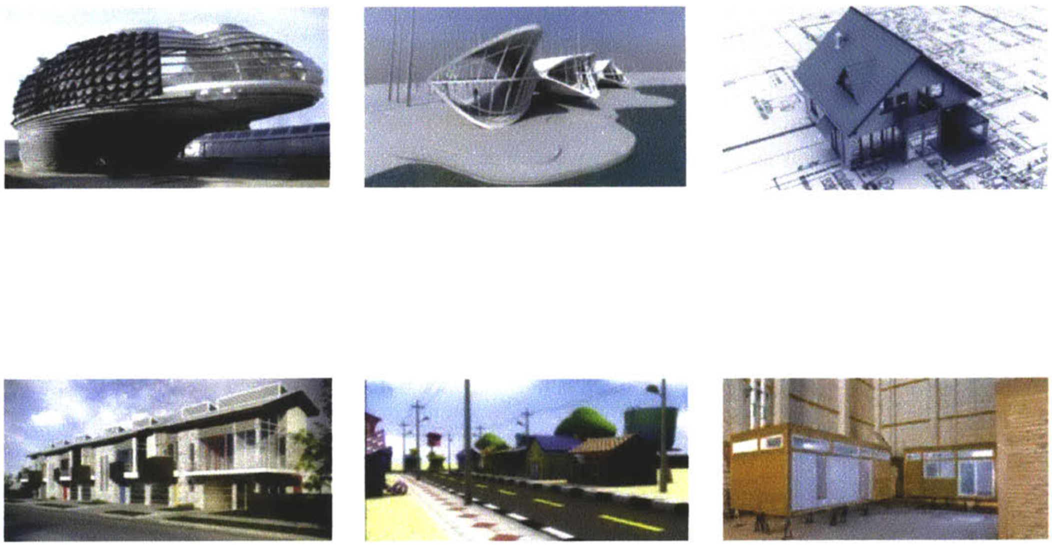


Figure 2-6: Six houses designed using mainstream CAD, where "if you just look at the work of many young designers, you can immediately see what sort of CAD system they are using" [Mitchell, 2009]

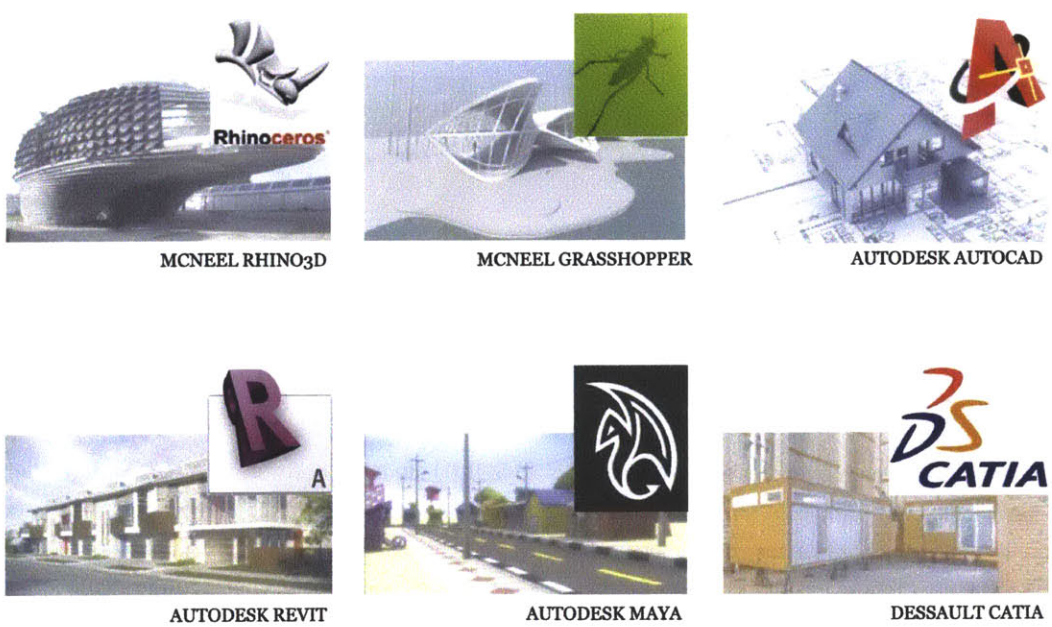


Figure 2-7: The CAD software packages in which this 6 houses in Figure 2-6 where designed.

We can sample six house designs modeled with the most common CAD packages employed in the architecture studio and practice (Figure 2-6). Going back to Mitchell's quote, we can easily identify what CAD software company is behind these representations (Figure 2-7) because of the very nature of their geometric kernel. There is a certain "style" in each of these examples coming from the main purpose of the tools that assisted their creation (Figure 2-8).

For example, the first case, has been built with McNeil's Rhinoceros3d modeler that uses non-uniform rational basis spline (NURBS) as its main geometric description kernel. The development of NURBS began in the 1950s by engineers who were in need of a mathematically precise representation of freeform surfaces like those used for ship hulls, aerospace exterior surfaces, and car bodies in order to improve the current design methods of single physical models. The pioneers of this development were Pierre Bzier who worked as an engineer at Renault, and Paul de Casteljau who worked at Citron, both in automation companies. At first NURBS were only used in the proprietary CAD packages of car companies. Later they became part of standard computer graphics packages, like architectural design ones. The user interface of a design program and its original target market, can have as much of an impact on our ability to get the job done as which underlying mathematical technique was used to define the objects. NURBs are the best technique available for smooth curve and surface design of "organic" house shapes like the ones in cars or ships, but we can only get the most out of a 3D modeling software, if we better understand its mathematical capabilities.

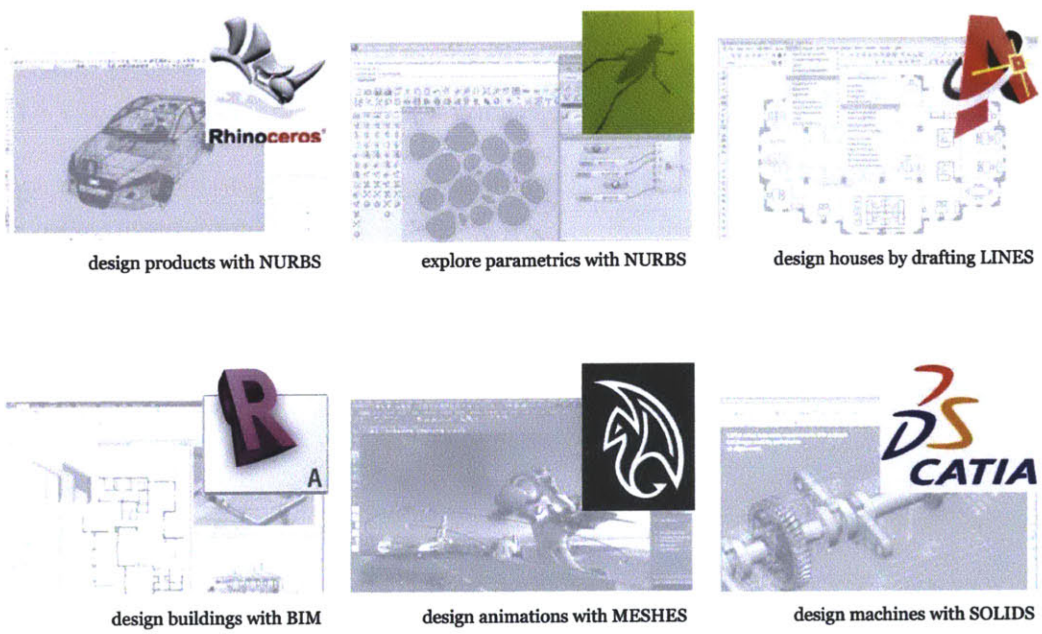


Figure 2-8: The purpose for which CAD tools were originally created may take over the designer's intent due to the rigid geometric assumptions running behind the scenes.

The last case has been built with Dessault Systems's 3d SOLID modeling software; CATIA. Solid modeling is a consistent set of principles for mathematical and computer modeling of three-dimensional solids and is distinguished from related areas of geometric modeling and computer graphics by its emphasis on physical fidelity. The use of these technique allows for the automation of several difficult engineering calculations that are carried out as a part of the design process, such as simulation, planning, and verification of machining and assembly. CATIA (Computer Aided Three-dimensional Interactive Application) is a software package that enables the creation of 3D parts, from 3D sketches, sheet metal, composites, molded, forged or tooling parts up to the definition of mechanical assemblies. It provides tools to complete product definition, including functional tolerances, as well as kinematics definition. For architects, that means that designing with CATIA will obey the logic of solid modeling and can efficiently target mass production of prefabricated houses, as the ones in the example.

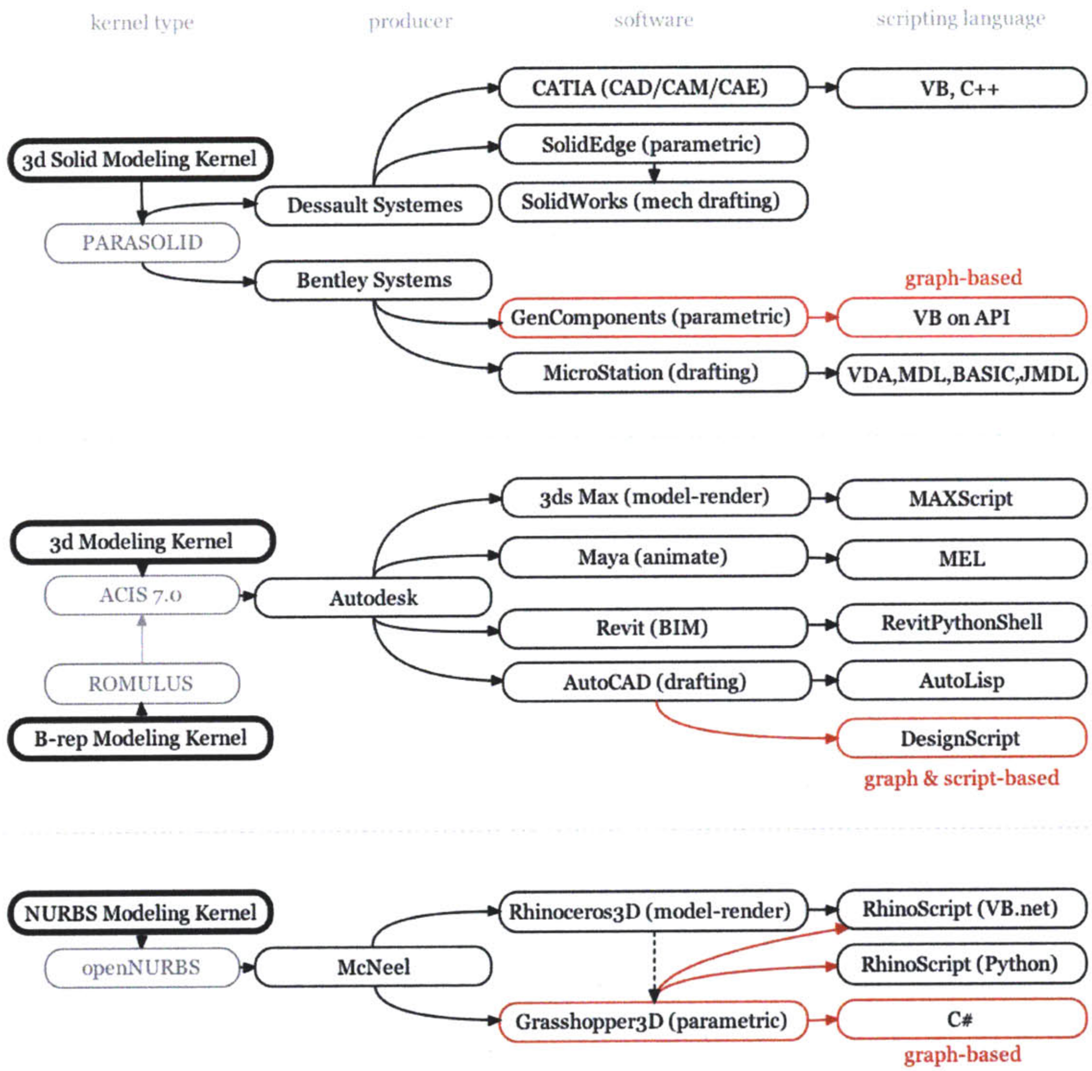


Figure 2-9: Diagram depicting Geometric Kernels behind most currently used computer-aided design tools.

Geometric kernels – based on geometric modeling conventions such as SOLIDS, B-REPS or NURBS – are also responsible for the available functionsand libraries that we can access when scripting or using parametric tools over CAD software packages. So we must understand that even generative scripting will be subedited to the underlying geometric kernel, and by getting to know its nature, we will better know what, the tools we use, are capable of. In Figure 2-9 we can see which are the geometric kernels behind the most used computer-aided design tools in practice, research and education, and how the programming library serving the CAD user interface, is the same accessed by any scripting or visual programming capabilities of the software platform. So generative scripting is also subject to the constraints of the CAD platform provider. This is one of the reasons why some explorations involving conflicting constraints that challenge imagination are better investigated in stand-alone applets that closely tailor rule systems and phenomena rather than geometric operations.

2.1.2 Computation and Computerization

In design, the main challenge of computation does not lie in mastering Computational Design[5] techniques, but rather in "acculturating a mode of computational design thinking" [Menges and Ahlquist, 2012]. If we look at the definition of computation, we can envision a different way to be with the computer, to think about the computer aided design process. The difference between computation and computerization is the same as the difference between deducing and compiling, the first process "increases the amount and specificity of information, while the other only contains as much information as is initially supplied [Terzidis, 2006][Menges and Ahlquist, 2012]. This means that a computational approach enables designs to be released from abstraction and organization into a deductive process of changing values and actions, as opposed to the computer-aided approach where there is a hierarchical process of describing objects as symbolic representations.

But computer-aided design was not always a synonym for computerization.

As a matter of fact, the early experiments on computer-aided tools where more similar to the definition of computation explained above. Sutherland's Sketchpad[6], developed at MIT, described as a man actually talking to a computer, allowed for variation and design formation to occur within a process (Figure 2-10 and Figure 2-11). Sutherland was able to connect the display capabilities of the CRT, with the computational abilities of the computer, and the interactive process made possible with the light pen [Carlson, 2007].

Figure 2-10: 1963 Ivan Sutherland at MIT designing with Sketchpad in a process driven by the designer.

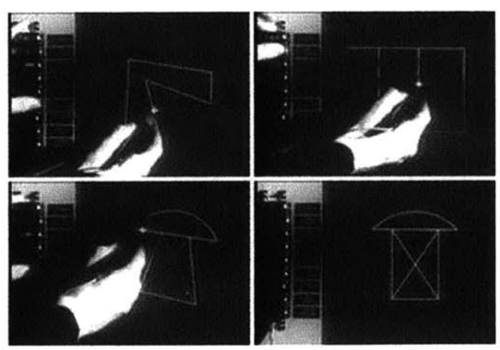


Figure 2-11: Starting with elemental properties and generative rules to which the computer responded with formal interpretations.

In Sketchpad, the drawing process was driven by the designer starting with elemental properties and generative rules and deriving some form at every step [Menges and Ahlquist, 2012]. The computer interpreted

[5]The process of creating computer programs as part of the design process, going beyond using them as a tool, but as a medium. It is a mode of precisely articulating design needs and principles to then use the computer's calculation power to explore a multiplicity of possible solutions in the design space. An approach referred in a Descartes' Dictum: *how can a designer build a device which outperforms the designer's specifications?* [Cariani, 1991]

[6]Sketchpad was a revolutionary computer program written by Ivan Sutherland in 1963 in the course of his PhD thesis, for which he received the Turing Award in 1988, and the Kyoto Prize in 2012. It helped change the way people interact with computers.

the designer light-pen gestures on the screen into lines, trims, arcs and relationships in the evolving shape (Figure 2-12 and Figure 2-13).

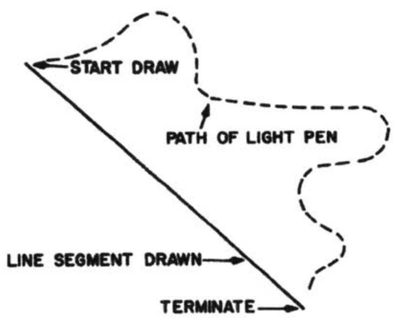


Figure 2-12: Sketchpad steps for drawing straight lines. "If we point the light pen at the display system and press a button called draw, the computer will construct a straight line segment which stretches like a rubber band from the initial to the present location of the pen" [Sutherland, 1963]

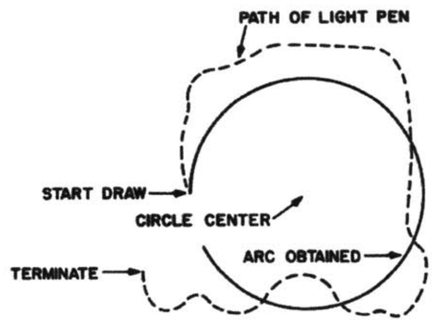


Figure 2-13: Demonstration of Sketchpad steps for drawing circle arcs.

This approach to software design tools envisioned the possibility of feedback between man and computer, but unfortunately did not persist in practice, and soon CAD became merely a product with prescribed operations that the designer used, rather than collaborated with. So, how do computers aid the designer today?

2.1.3 Current Computational Aids

In the history of design and design theory it is sometimes believed that design software is no more than a channel through which the designer's ideas are carried along, landing in the world of materiality and description [Cardoso, 2010]. As if the machine were a passive device with no effect on the design process. Nevertheless every designer has a computer that he uses more than eight hours every day, and that is filled with design software packages described as performing predicted tasks. What if our design problem is outside of the box of predicted tasks? Assuming that the computer is the designer's main work medium, there is an emerging tendency to use it as a tool in the early creative process.

Parametric[7] and Scripting[8] add-ons to standard computer aided design packages enable access to the CAD solver's[9] geometric backbones, and in consequence designs produced with this aids tend to focus on generative descriptions, rather than on result representations. However, today's designers are yet not as agile in scripting processes as they are in representation tools, so we can still perceive some "style" in the results of scripting within CAD (Figure 2-14, Figure 2-15). This is manly due to the formal descriptions of the geometrical operations described in the software core, and also to the fact that "fashionable" algorithms shared in online forums satisfy unexperienced users that apply them with little forward invention.

It is nonetheless true that architects use scripting because that have the need to see differently and take advantage of the computer's potential.

[7] It is the "lowest level of active computing, instead of defining fixed geometry the designer sets variable geometry and construct alternative rule-driven forms as parameters are rearranged in real time. It is a very extended technique in the aerospace industry for the design of new components by making small modifications to existing ones so that the new performance will be easy to predict." [Frazer, 2005].

[8] Scripting enables people who are not professional developers to modify a software artifact within its prescribed kernel constraints.

[9] A solver is a generic term indicating a piece of mathematical software, possibly in the form of a stand-alone computer program or as a software library, that 'solves' a mathematical problem. A solver takes problem descriptions in some sort of generic form and calculate their solution. In a solver, the emphasis is on creating a program or library that can easily be applied to other problems of similar type.

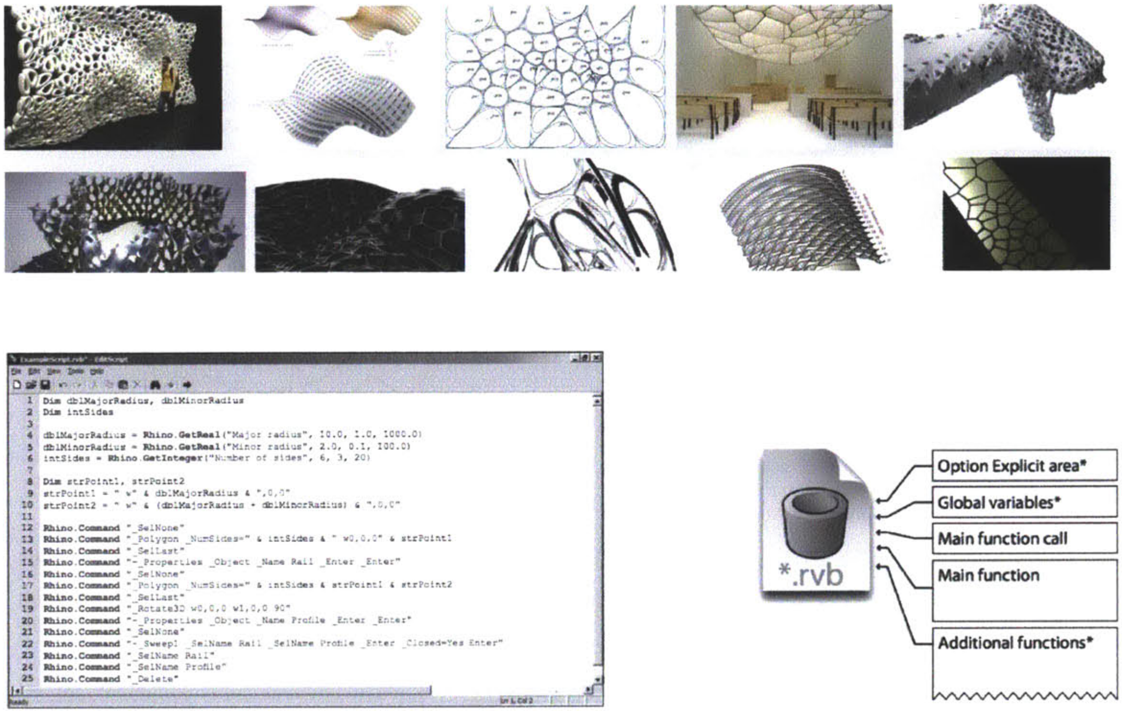


Figure 2-14: Scripting environment within McNeel's Rhinoceros3d modeler using RhinoScript.

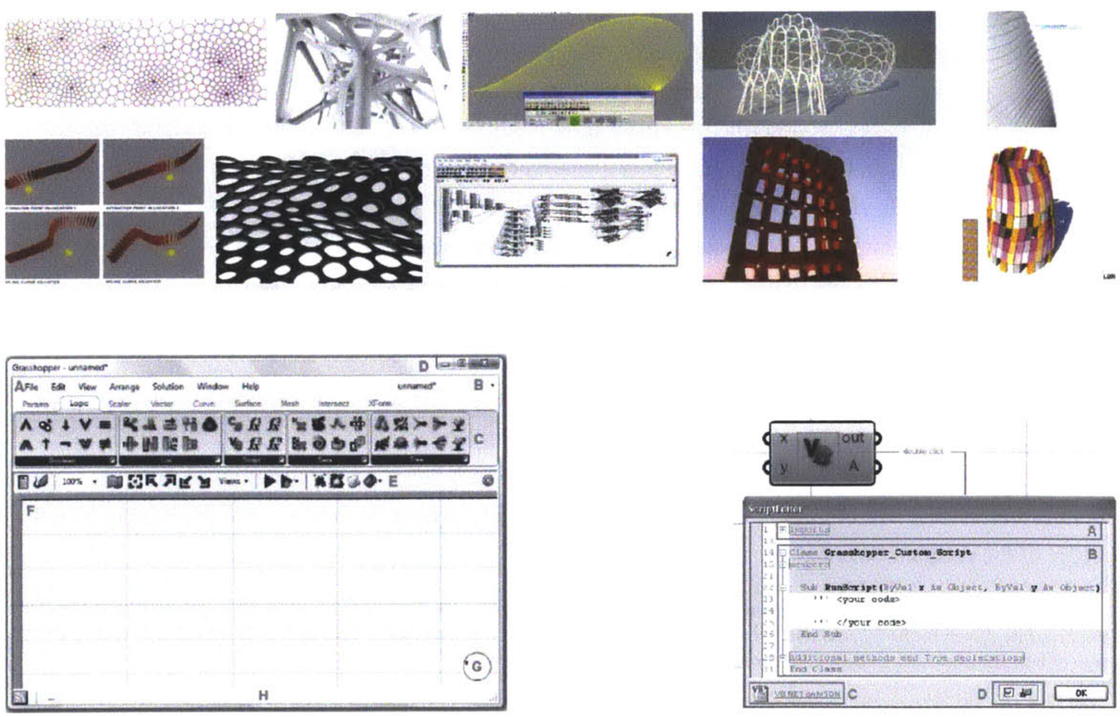


Figure 2-15: Scripting and Visual Programming environment within McNeel's Rhinoceros3d modeler using RhinoCommons library through different programming languages.

Robert Aish[10], in one of his latest papers explains how future Autodesk Research is looking into the possibility to provide both imperative (rigorous scripting like RhinoScript or AutoLisp) and associative scripting (based on visual graph programming like Grasshopper) capabilities within CAD.

Robert Aish defines DesignScript as new Autodesk programming language that goes beyond the limitations of the 2d drafting and the BIM eras, expanding the accuracy from model to reality. But, for the designer wishing to experiment, can it be more successful to escape the CAD platform problem-solving and analysis targets, and enter the DIY tools era? (Figure 2-16)

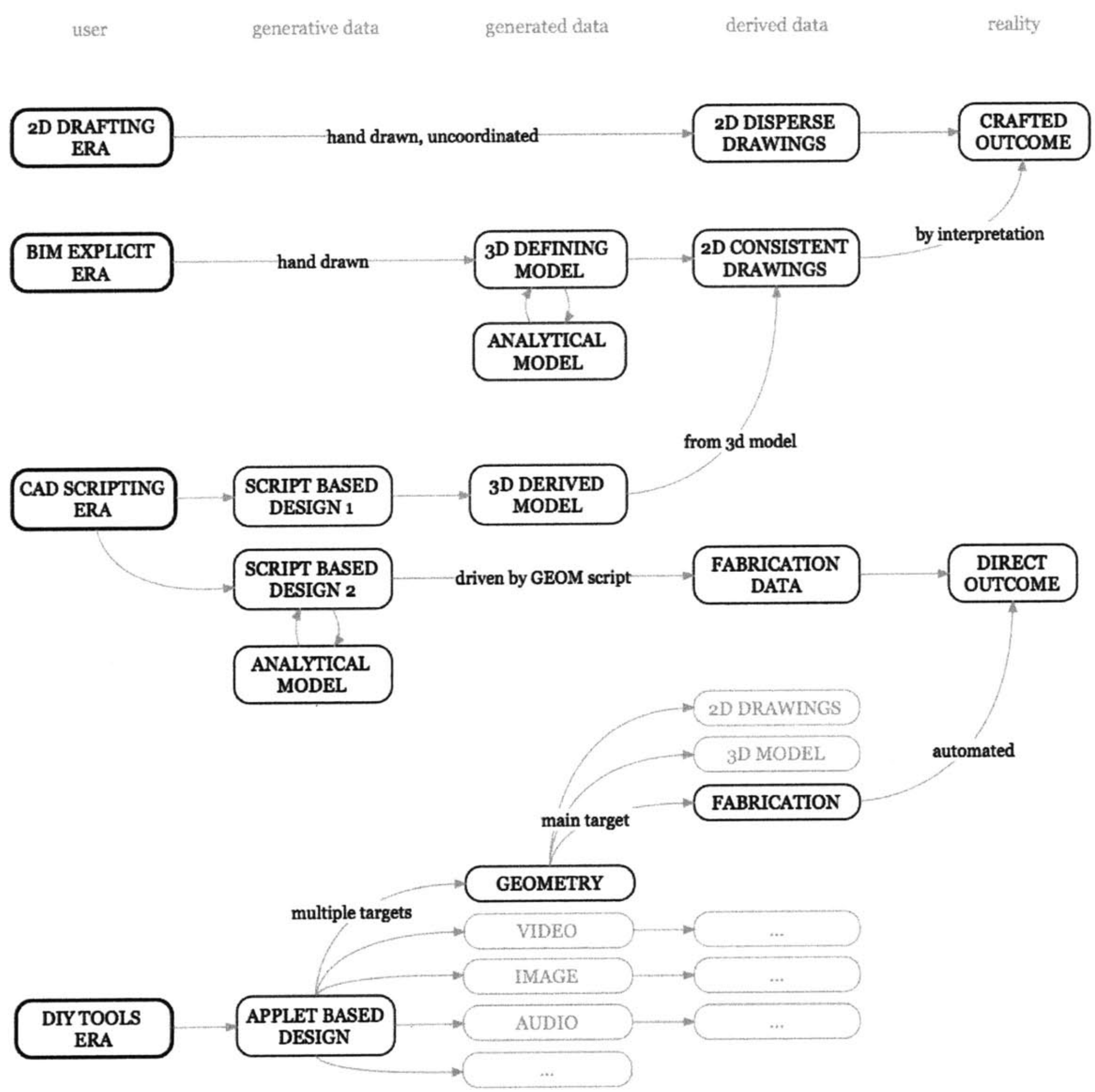


Figure 2-16: DesignScript as new Autodesk programming language that goes beyond the limitations of the 2d drafting and the BIM eras (Adapted from Robert Aish talk "DesignScript, a new programming language for designers", 2013, Harvard GSD Forward Talks Lecture Series).

[10] Robert Aish studied Industrial Design at the Royal College of Art in London and has a Ph.D. in Human Computer Interaction from the University of Essex. He has developed engineering software with Arup, architectural software with Rucaps, naval architecture software with Intergraph and the GenerativeComponents parametric design software with Bentley. In 2006 he received the Association for Computer-Aided Design in Architecture (ACADIA) Society Award.

Aish claims that associative scripting – dedicated to form dependencies – hides the programming language behind a user interface of components and connections, that does not scale to complex projects. Also more accomplished users, that have the need to use imperative scripting, do not find accurate ways to integrate the code to the associative graph structure. He also addresses the fact that imperative scripting is overwhelming to novice users and that the first steps into computational thinking are much easily taken through associative scripting. He then proposes a new scripting language, "DesignScript" running in Autodesk software packages like Autocad, that integrates both imperative and associative logics so that the user can switch between them when needed. "DesignScript can be viewed as part of the continuing tradition of the development of parametric and associative modeling tools for advanced architectural design and building engineering". "How can a computational tools invoke a computational mindset and in turn contribute to design thinking? Using DesignScript is a new way of designing with its own expressive possibilities. But there is a level of understanding required to harness this expressiveness and this suggests a level of rigor and discipline. The argument is that the experience of learning and using DesignScript contributes not just to the expressiveness and clarity of the resulting design but also to the skills and knowledge of the user. In short,a new toolset suggests a new mindset." [Aish, 2011].

However, in the eyes of an experienced user, scripting and parametric plug-ins embedded in standard CAD, soon also reveal limitations related to the impossibility of accessing certain geometric modules; the enforced use of a set of libraries[11]; and the underlying rigid geometric history. Even if we use imperative or associative scripting techniques, the platform in which we operate can have a limiting set of constraints and the hierarchical structure.

2.2 Reconquering Tools

Computation has a profound impact on both the perception and realization of architectural form, space and structure. It shifts the way one perceives form, the way in which form is purposed, and the way in which form is produced.
–Achim Menges "Computational Design Thinking", 2012

Computation behind CAD does have a profound impact in everyday design, it shapes our imagination through the use of representation conventions that someone has programmed. But it is the understanding of computation that can also help us investigate new rules. Digital design can be understood as the search for structures of originality, for designs beyond the imagination, with a complexity and dimensionality exceeding traditional techniques. Today's design process relies on standard computer aids that are mainly focused on geometrical representation. But do we only design with geometry? It forces designers to make an extra effort to incorporate these structures of originality into the design process.
More and more designers learn scripting within CAD environments, so building on this ongoing tendency, independent self-made computational mechanisms that the designer itself writes, revisits and integrates, can be even more beneficial to design explorations.

[11]In computer science, a library is a collection of behavior implementations in a programming language, that has a well-defined interface by which the behavior is invoked.

2.2.1 The Tool to Think with

For Paul Coates, one of the pioneers in algorithmic design[12], the computer can make a difference in design research when it is used not only as a drafting or modeling tool, but as a tool to think with, like the Turtle in LOGO is a constructed computational "object- to-think-with" in the Papertian Constructionist way" [Coates, 2010]. "The Turtle is a computer-controlled cybernetic animal. It exists within the cognitive mini-cultures of the 'LOGO environment', LOGO being the computer language in which communication with the Turtle takes place. The idea of programming is introduced through the metaphor of teaching the Turtle a new word. The Turtle serves no other purpose than of being good to program and good to think with [Papert, 1980].

This approach to design leads to an active engagement with the machine. Coates clarifies the difference between *active* and *passive* computer use; active use is that of representing architectural designs as code scripts generating emergent outcomes, while passive use is that of representing them as static geometry [Coates, 2010]. He emphasizes the idea that representational tools like CAD should be pushed back in the design process until the design exploration is complete (Figure 2-17).

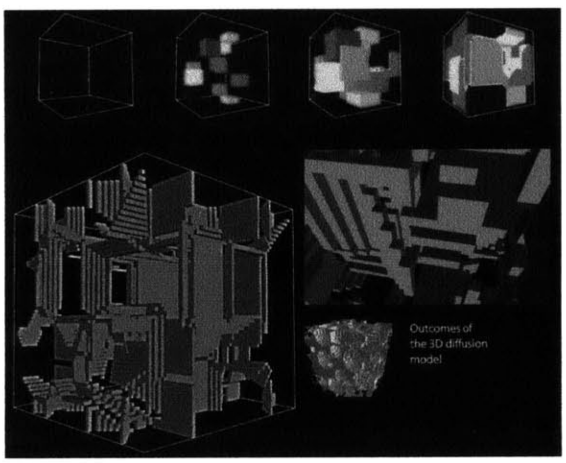


Figure 2-17: Reaction diffusion algorithm results. "A series of emergent outcomes which are the result of an extremely simple algorithm. The complexity comes from the many different ways that the expanding cubes intersect. This effect is achieved by simply focusing on the process rather than trying to calculate the intersections in some top-down method" [Coates, 2010].

Coates also talks about the "cascade of observations" to the outcome of a program. He describes the "Top as person looking at the computer using brain/eye looking at: the Global observer/reporter in the program computer observer of: Local agents in the program who just observe their immediate environment" [Coates, 2010]. In the case of the designer as tool-user of software, this is as far as it goes. If the designer is also the programmer, then the whole process is put into a loop where the person can actively change the way the various observations work on the basis of some goal.

The same differentiation between active and passive, can be perceived in the work John Frazer, a contemporary of Coates and pioneer in computation design education, when he describes *active computing* environments as environments where you do not do the same thing twice, and *passive computing* environments as mostly what architects do in CAD, automated drafting processes [Frazer, 2005]. Frazer is interested in the nature of evolutionary computer models, he argues that in describing the models, "it is recommended

[12]As claimed by Mark Burry's computational design survey in his recent book "Scripting Cultures".

that the concept is process-driven: that is, by form-generating rules which consist not of components, but of processes" (Figure 2-18).

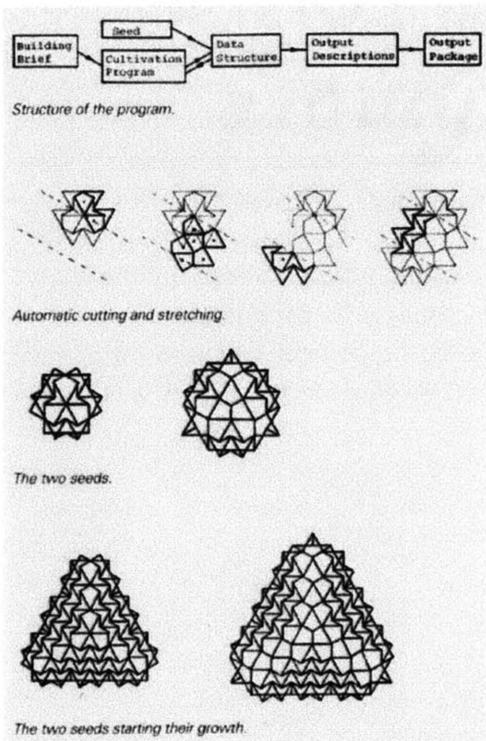


Figure 2-18: "The basic data structure was set up and manipulated by a series of machine-code sub-routines and functions. These allowed the descriptions of the units to be retrieved, deleted or updated, and additional units to be inserted into the chain". "A variety of descriptions were available depending on the detail required in the final drawing. The GINO graphics package was used to produce perspectives and other drawings required" [Frazer, 2005].

Both Coates and Frazer see computers as creative tools for learning, they believe in the need to encourage abstract thought rather than learning the standard procedures within tools. They inherit some of the first artificial intelligence pioneers idea that the computer allows a new way of thinking about knowledge, and it is not just a more powerful calculator or data processor [Coates, 2010].

> *You have to distinguish between writing a program that helps you test your theory or analyze your results, and writing a program that is your theory.*
> *–Marvin Minsky "The Society of Mind"*

Marvin Minsky[13] argues for "writing a program that is your theory". We can easily read here 'theory' as 'design' and say that the self-made applet is the design, going back to Pangaro's research on "designing the designing" in a conversation. Today, when CAD has become friendly enough to make us expert users, can we reconquer design tools and become eager toolmakers? Can we use applets to plant design seeds? To explore the design space? To think with?

[13]In the early 1970s at the MIT Artificial Intelligence Lab, Marvin Minsky and Seymour Papert started developing what came to be called The Society of Mind theory. The theory attempts to explain how what we call intelligence could be a product of the interaction of non-intelligent parts.

2.2.2 Not a CAD alternative

We can see applets not as a CAD substitute – because their nature and purpose is not drafting – but as contributors to the early design process. Paul Coates remarks the following when talking about writing simple rules that develop an emergent distribution into varieties of forms; "after everything has settled down, the emergent consensus, the self-organized turtle configurations can be exported to other packages for the further processing. For example the turtle coordinates are read into AutoCAD using a small Visual Basic script. Further processing to tile up the mesh and rendering can be achieved with your favorite CAD package" [Coates, 2010]. Self-made computation is an enabler for computational design thinking, a provider of a way to think and create with the computer that can then be combined with all the other representation, collaboration and evaluation tools that inhabit current design processes.

In this thesis I argue that we can envision self-made environments that interrogate design computation. Building on the designer's current emerging scripting agility in digital design, small tailor-made software applets can be appropriate companions in specific project development moments. For those problems that find CAD too generalist or too restrictive, we need to build a space in which to establish our own design rules, in which to see differently, with no hierarchical representation behind the scenes, driving us through constraint descriptions where we miss the opportunity to invent with the computer's assistance.

2.3 Design Workflows

What is the design workflow today and where do applets fit? By exploring past and present design models and design workflows we see that there has been a shift from problem-solving to problem-finding and that applet-making is contributing to put more emphasis on that. I do not pretend to elaborate here a comprehensive historical survey of how design methods have been described by scholars and practitioners, as models of the design process became refuted very fast and succeeded one another without ever finding the model that describes what we do when we create. Instead I use some of these models to support the idea of a shift.

> *Neither the human purposes nor the architect's method are fully known in advance. Consequently, if this interpretation of the architectural problem situation is accepted, any problem-solving technique that relies on explicit problem definition, on distinct goal orientation, on data collection, or even on non-adaptive algorithms will distort the design process and the human purposes involved.*
> *– Stanford Anderson, "Problem-Solving and Problem-Worrying", 1966*

Back in 1966 Stanford Anderson[14] gave a talk at the AA[15] entitled "Problem Solving = Problem Worrying" which argued against the then fashionable notion that the design of a building could be done as a linear rational enterprise, "starting from the problem definition and ending in the solution". He suggested that the process was more likely to be one of "problem worrying", "where a cycle of problem definition, partial solution and redefinition was always necessary". He claimed that that is because it is impossible to define all the

[14] "Stanford Anderson, an architectural theorist who wrote a seminal paper in 1966 on computer-aided design, decrying the box-ticking reductionist approach which he called the justification approach – when after a lengthy brief constructing process by the "expert client", a building was said to be fully functional when all the aspects in the brief could be checked off. Instead he proposed problem worrying, where a cycle of problem definition, partial solution and redefinition was always necessary, as opposed to problem solving" [Coates, 2010].

[15] The Architectural Association School of Architecture in London, commonly referred to as the AA, is the oldest independent school of architecture in the UK and one of the most prestigious and competitive in the world. Its wide-ranging program of exhibitions, lectures, symposia and publications have given it a central position in global discussions and developments within contemporary architectural culture.

problems that an architectural object has to solve; and because the production of an actual morphology feeds back on the initial problem statement because any proposal will entail originally unintended consequences [Coates, 2010]. So, instead of assuming that every problem can be identified, and thus can be solved; the emphasis lays on the understanding of it in a continuous "worry" and the construction of the environment, to support the success of design purposes, that allows for reformulation. This idea of a constant reformulation capability is also a main goal in Shape Grammars where, even if invariants exist, the system needs to allow for discontinuities at any time because we cannot predict when we will need to see differently [Stiny, 2006]. These discontinuities can be approached in the applet-making culture as rule changes and adaptations to new needs and constraints of the project at hand.

There are as many "design processes" as designers and those processes cannot be captured in staged problem-solving. Instead problem-finding or problem-worrying describe much more exploratory ways closer to how design works. When designing we look for good approaches, all with different structures and constraint weights, that then we integrate and evaluate without an optimal target in mind. In 1987, Peter Rowe explains that his book, "Design Thinking", "is a neutral account, an attempt to fashion a generalized portrait of design thinking", he is concerned with "the situational logic and the decision-making process of designers in action, as well as with theoretical dimensions that both account for and inform this kind of undertaking". As Stiny and Anderson, Rowe argues that "the" design process does not exist "in the restricted sense of an ideal step-by-step technique, rather there are many styles of decision making". He also presents case studies of designers in action where he aims to "reconstruct the sequence of steps, moves, and other logical procedures that were employed," and he calls this reconstruction a "protocol" [Rowe, 1987].

Applet-Making encourages broad exploration in the early design conception to avoid what Rowe encounters after having explored the protocols. He observes the "dominant influence that is exerted by initial design ideas on subsequent problem-solving directions" that makes designers inevitably try to make the initial idea work even if severe problems are encountered, rather than "stand back and adopt a fresh point of departure". He also states that "the "style" of the completed projects seems to have been determined primarily by two factors"; initial ideas and "the sequence in which design principles where applied seems to have mattered the most. Contrary to some earlier so-called design methodologists, the kind of theory we need if we are to explain what is going on when we design must go beyond matters of procedure" [Rowe, 1987].

To evidence these shift from staged design models to non-standard explorative ones, we can observe how from Asimov in 1962 (Figure 2-19) to Archer in 1964 (Figure 2-20), we already start to see a change from a very staged deterministic convergent description of design thinking, into something with fuzzier parts, and more importantly, feedback loops to the early stages.

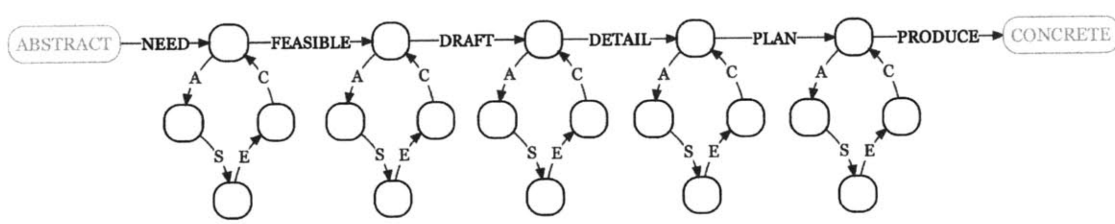


Figure 2-19: Decision making cycle at every design phase in a problem-solving defined fashion (Asimow, 1962, A-S-E-C: Analysis-Synthesis-Evaluation-Communication).

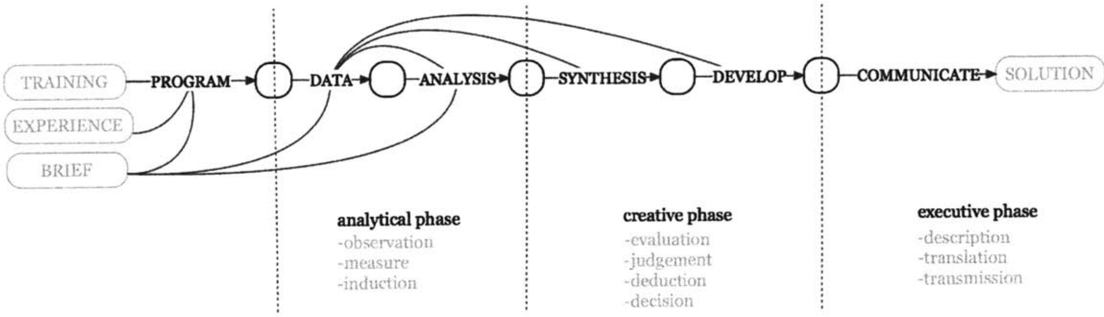


Figure 2-20: Model of stages of the design process (Archer, 1964), where feedback loops are more present than in earlier models.

2.3.1 From Problem-Solving to Problem-Finding

Early theoretical positions on design methods did not confront the "wicked problems" of design [Bazjanac, 1975] that are those not having a definitive formulation, open-ended by nature, their solution depends on the initial definitions, can always be reformulated, and have no ideal solution. Still in 1994 the "Architect's Handbook of Professional Practice" claims for three stages in design; Schematics, Development and Documentation (Figure 2-21). Where the first phase is the "creative" one, even if creativity is a constant from start to end in architecture. Most of the computer-aids target the second and third phases, where solutions have already been chosen, where there is no more explorations to make. Can we tailor other tools that can address the, so called, "creative phase" and extend it as much as possible?

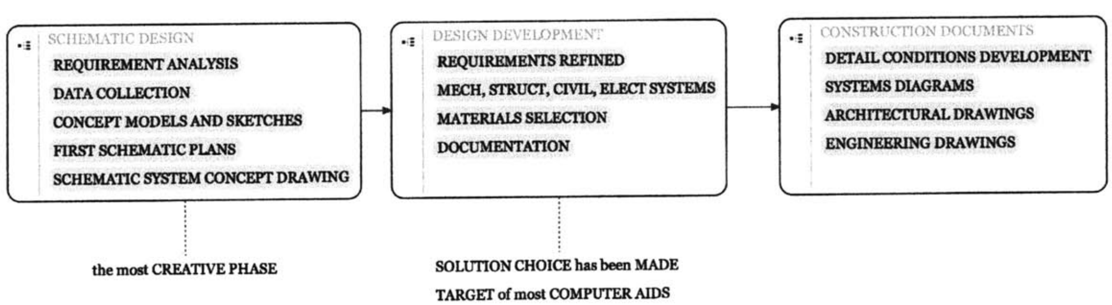


Figure 2-21: Division of design activity into phases, adapted from "The Architect's Handbook of Professional Practice" (Cryer, 1994, p. 526). Most computer-aids to design "operate in the Design Development Phase, where a solution has already been chosen and is refined through analysis" and there is no further creative iteration [von Buelow, 2007].

As Peter von Buelow [16] explains, "in recent decades millions of lines of programming code have been directed at providing designers with help in the technical or knowledge based side of their activity, with almost no successful attempts to provide aid to the creative aspects of design as well." [von Buelow, 2007]. He establishes the analogy that the designer tends to see the computer as the "brain" and that he takes the role of "apprentice – posing questions to the master (computer), and respectfully awaiting the answer". Can we reverse that? Can the computer become our apprentice?

"Computer tools used in conceptual design should, and can, exhibit the quality of creative stimulation" [von

[16] Dr.-Ing. Peter von Buelow has worked as both architect and engineer, and is currently a professor at the University of Michigan in Ann Arbor USA, where he teaches structures in the School of Architecture, and conducts research in structural form exploration based on evolutionary computation. In his thesis from 2007, he explores design tools based on evolutionary computation (EC), oriented primarily toward conceptual design of architectural and civil engineering structures.

Buelow, 2007]. For his thesis proposal, Buelow develops an "intelligent genetic tool" for the designer to use. One that will incorporate the three attributes that he defines as "describing the design activity"; purposeful, goal oriented and creative. He argues that any aid to the design process must also respond to these aspects of the activity. I argue that the tool needs to be made by – and not for – the designer, and that it can come from genetic or from other kinds of computation, the type of computation at play should be the designer's choice according to every project he faces.

In the sense of Buelow's assumption that design has purpose, goal and creativity, Christopher Alexander points out that, although design itself may be "contrived in the mind", "the ultimate objective of design is form" [Alexander, 1966]. He sees design as the "fit" between form and context. I argue that there are many paths to get to "form" with the computer, many contexts to work with other than those offered by current CAD tools. Computational rule-making, or applet-making, is an activity that enables collaboration and discovery with the computer to get to form. We discover by letting the machine be our apprentice who makes proposals that the designer critiques and integrates, in the way fresh provocative questions enable the most experienced to see differently.

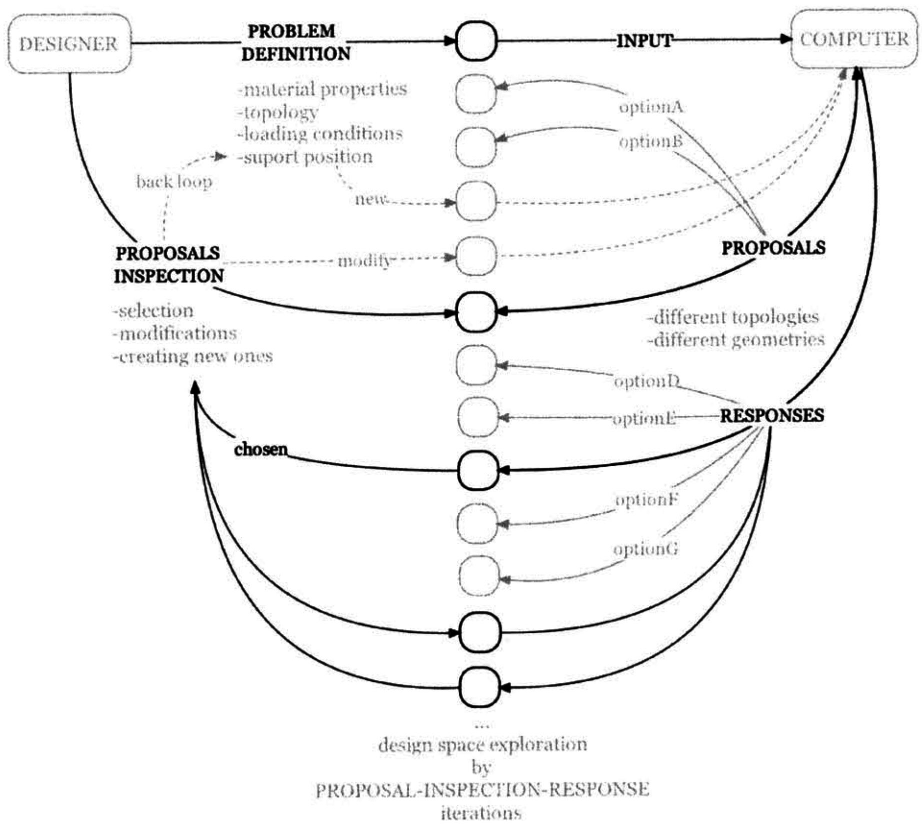


Figure 2-22: Diagram of the overall conversation adapted from Buelow's text, where the proposed design tool interacts "intelligently" with the designer. [von Buelow, 2007].

In Buelow's research on generative design tools, there exists a conversation between the designer and the device where one integrates, selects and describes and the other one proposes trying to maximize the feedback in the process and the possibility to go back and change the rules, the problem definition, or the introduction of new constraints and parameters for structural design exploration (Figure 2-22).

Then in 2009, Holzer contrasts the traditional model (Figure 2-23) where experts give feedback after the main decisions are made, and that of optioneering (Figure 2-24) where architect and experts, together, come up with multiple solution generation. These enables the incorporation of the maximum design constraints, or rules, at the beginning of the process, but it does not explain how to make a generative process work.

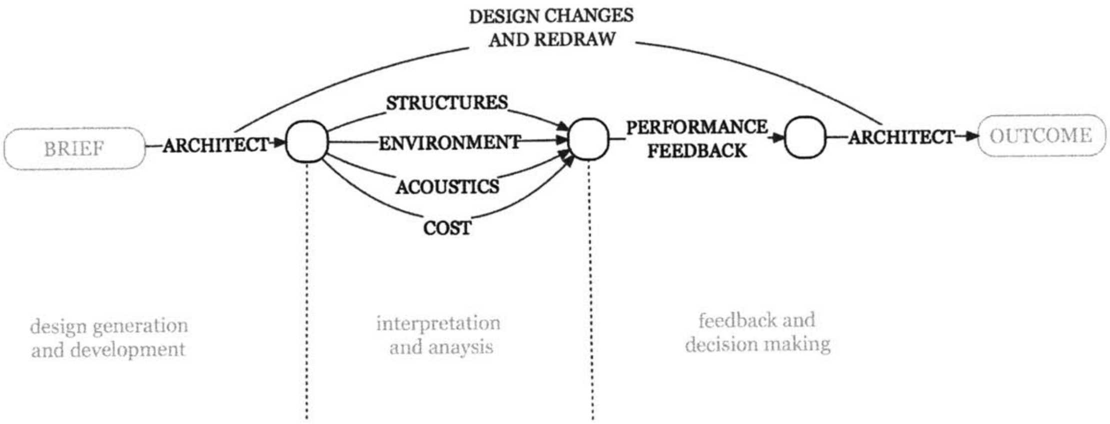


Figure 2-23: Traditional workflow where the generation of design options and the interpretation and analysis occur in separate steps (1995), adapted by the author.

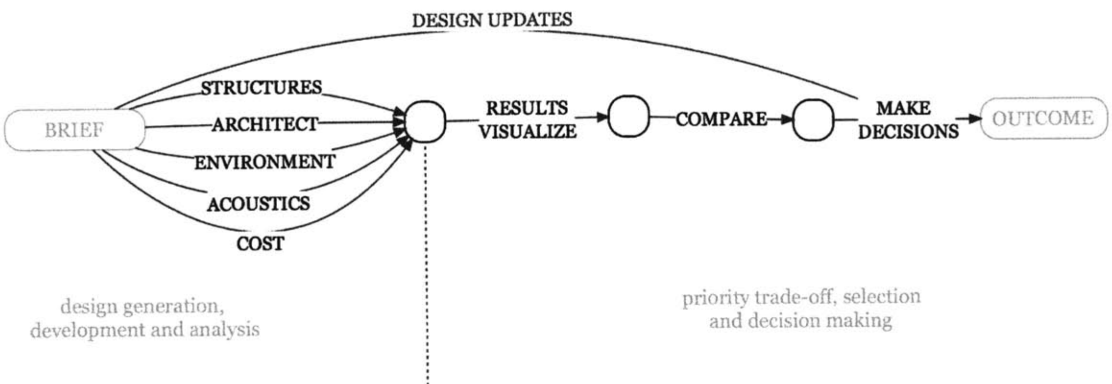


Figure 2-24: Optioneering approach where design options is linked to early design, enabling multiple solution generation (2009), adapted by the author.

This thesis addresses ways to seek feedback between designer and medium, through having a conversation with the machine, a bidirectional process of invention. Other modern models like the IDEO one (Figure 2-25), or the Stanford D-school one (Figure 2-26), focus on the innovative part of design thinking, a model that Paul Pangaro[17], cybernetics of language researcher and conversation theorist, attempts to counter-propose. He claims that problem-finding and designing for conversations is much more appropriate for wicked problems (Figure 2-27).

[17]Paul Pangaro is a technology executive, conversation theorist, entrepreneur, and performer. He combines technical depth, marketing and business acumen, and passion for designing products that serve the cognitive and social needs of human beings. He has worked as CTO for startups such as Idealab's Snap.com, developed product roadmaps for consumer Internet companies, and managed developer outreach and web properties for Sun Microsystems. From his background in the cybernetics of language, he has developed a methodology for modeling the necessary conversations for consumers to understand how products and services may be used to achieve their goals. He has collaborated closely with designers in creating high-traffic web sites and models of gnarly concepts such as innovation, play, and the creative process. From 2001 through 2007 he taught a class at Stanford University on the cybernetics of product design.

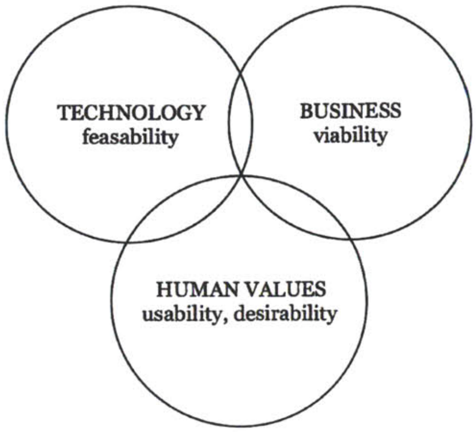


Figure 2-25: Design Innovation at the center of technology, business and human value (David Kelley, Tom Kelley at IDEO, 2000), adapted by the author.

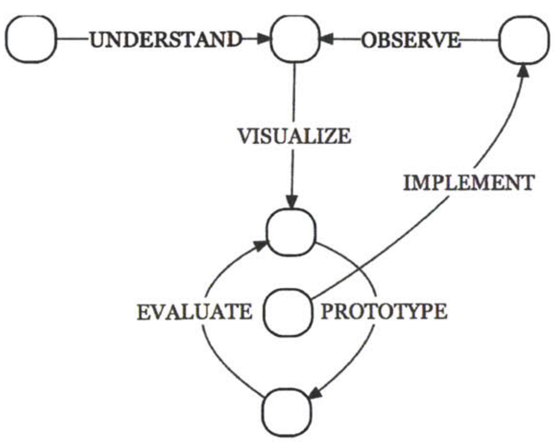


Figure 2-26: Standford's "d-school" design process model (Hasso Plattner, 2004), adapted by the author.

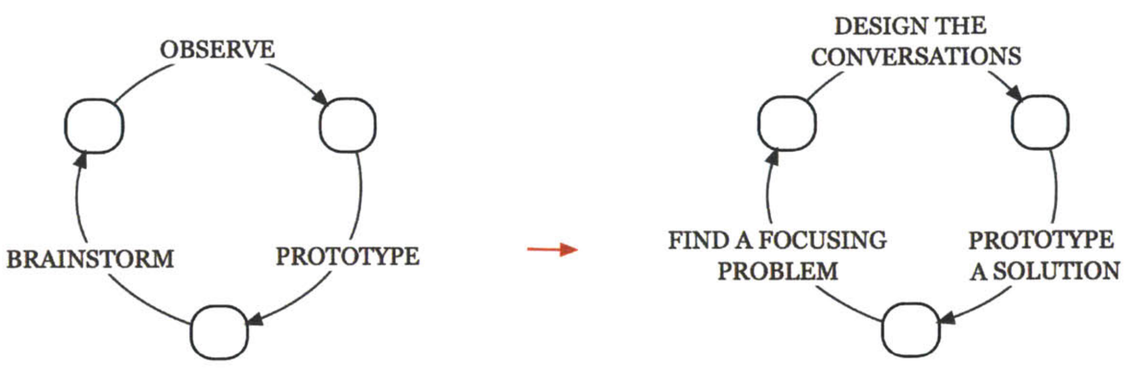


Figure 2-27: Paul Pangaro "rethinks design thinking and wicked problems" and formulates a counter-proposal (2010), adapted by the author.

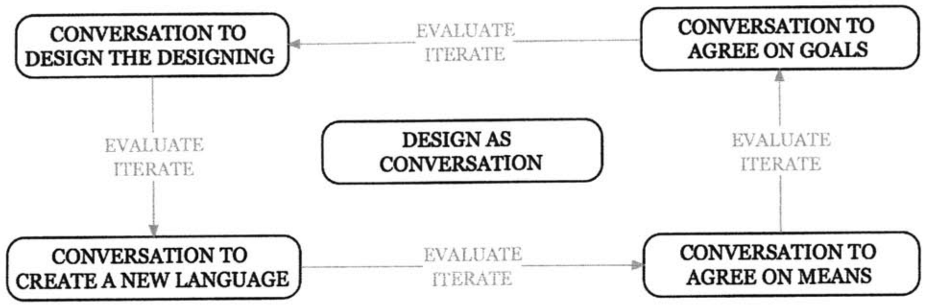


Figure 2-28: Paul Pangaro designing for a conversation, adapted by the author.

Pangaro offers a critique of "design thinking" grounded in a cybernetic perspective. He argues that conversations are the heart and substance of all design practice, and shows how a cadence of designed conversations is an effective means for us to comprehend, and perhaps even begin to tame, our wicked problems. He critiques "design thinking" as a formula, that includes brainstorming, prototyping, iterating and evaluating in a staged manner, and he proposes that we approach "design as a conversation", as a process of designing the designing, of problem seeking rather than step-by-step problem solving [Pangaro, 2011]. I argue that this is the same philosophy that self-made computation for design carries, where the designer builds his design tool to have a conversation with the computer; a conversation to agree on goals, means, create a new language and design the designing (Figure 2-28).

2.3.2 Workflows for Wicked Problems

Recently, Andrew Pressman[18] published a book with the title "Designing architecture, the elements of process", where different architects try to explain what the "design process" is, as Pressman describes, they try to "give voice to a multi-functional, even chaotic process" [Pressman, 2012].

> *Architects have many tools for visualization at their disposal, finding and using the right tool to fashion a design solution is a bit like woodworking - it takes practice. Drawing, model making, and digital media are not just methods to communicate an idea already formed - they are more valuable used as tools to explore design. Working on a drawing or a model is another form of conversation - between you and the design, back and forth. Suddenly you see something you didn't know was there.*
> *–Michael J. Crosbie, PhD., AIA*

> *Increasingly we recognize that the design process works with information and ideas simultaneously on many levels. Thus the architect can be thinking simultaneously about the overall geometry of the building, the ways a disabled person might experience the spaces, and the materials of which the building will be constructed.*
> *–Bradford Perkins, FAIA, Interview by Andrew Pressman, Albuquerque, NM, June 8 2000*

> *We are very question/answer oriented. Ours is a classical heuristic approach in which we attend to a process to arrive at a solution rather than to begin from a priori position. Many people think architects are solution oriented. The discussion we're having is actually about exploring*

[18] Andrew Pressman is an Architect and Professor Emeritus at the University of New Mexico and leads his own Architectural firm in Washington D.C.

a problem and articulating questions, which become the focus of our design investigations. The viability of the methodology is based on the question-posing process, which is an inclusive process designed to elicit creativity rather than to inhibit it.
–Thom Mayne, FAIA, Interview by Andrew Pressman, Albuquerque, NM, June 8 2000

What these reflections have in common is that there is no such thing as "the design process", there is no staged linear recipe to follow in architectural design. Maybe there are protocols and problem-worrying and conversations spanning different fields and rules. These rules may conflict with one another and may render the design target difficult to imagine. Little devices that the designer builds to investigate and think with can better assist in these explorative needs. Applets are in constant reformulation, are tools to design the designing and are tailored to be small enough that the designer is always in control of what they are made of and knows what is the formal assumption behind. More importantly, the rules are only described and can shift at every run, also new ones can emerge in a conversation with the computer.

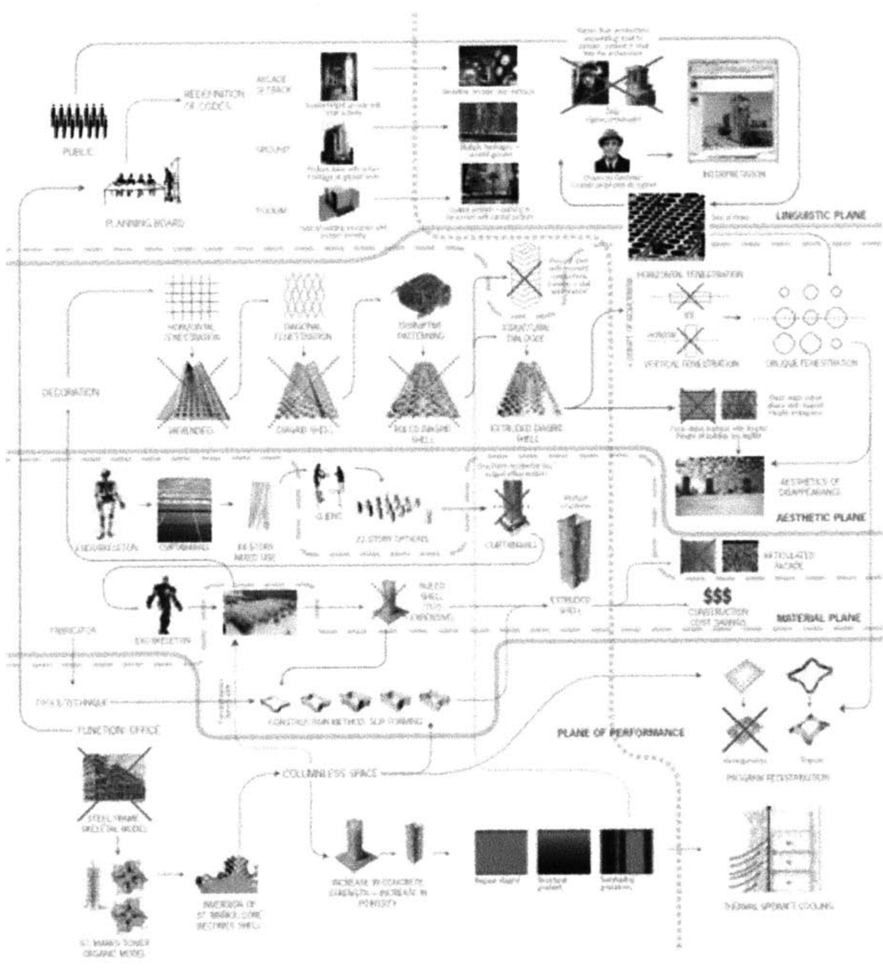


Figure 2-29: The design exploration for the 0-14 building. Combination of different tracks: Linguistic Aesthetic Material Performance (Diagram by Architects Reiser and Umemoto, 2010).

In wicked problems, the design process looks more like Reiser and Umemoto's one for the 0-14 building tower (Figure 2-29). In a recent article entitled "The scent of the System", they explain how the diagram tries to depict a combination of linguistic, material,performance,and aesthetictracks that combined brought a tower design into life. Reiser and Umemoto are suspicious about the role of computation in architecture, they claim that as a tool of analysis, computation can provide insight to better understand the nature of something that already exists, but as a tool of creation or origination, even with its daunting power of combinatory processing, it will at best merely mimic the subtlety of human thinking.

Actually, computation should not try to mimic design thinking. It should not take the role of the designer but be his apprentice, develop a relationship of back and forth propositions, from a described start point, that can expand the designer imagination. Co-authored inventions can arise tackling the problem-finding in many different ways, other than what we can see in convergent solution searches or computerizations. Going back to designing for conversations and Pangaro's "designing the designing", we see how we are moving from staged engineering problem-solving to constant feedback problem-seeking. However, in current CAD workflow models, the designer operates a software drafting tool by translating hand drawn decisions into digital language. The relationship between man and machine consists on a linear approach. The use of representation tools comes early in the process and there is late intervention of performance feedback (Figure 2-30), as we can also see in Holzer digram (Figure 2-23).

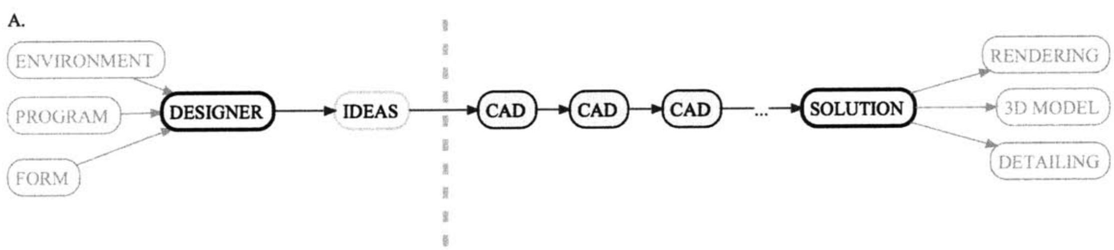


Figure 2-30: Contemporary design workflow models. [A] A linear drafting-based approach where representation tools come early in the process and there is late intervention of performance feedback.

In a parametric model, the geometry is a function of a finite set of parameters, and a collection of relationships between functions and parameters called schema. If the schema does not contain a parameter to modify the model in the desired way, the schema needs to be modified in a process that Woodbury describes as erase, edit, relate and repair [Woodbury et al., 2007]. Mark Burry in 1996, articulated one of the earliest instances of this problem in architectural practice, which occurred when he attempted to modify a schema of the Sagrada Famlia to generate paraboloids instead of conoids and concluded that there was no solution other than to completely disassemble the model and restart at that critical decision [Davis et al., 2011]. The rigidity of parametric models to unanticipated design changes (a common occurrence in a cyclic design process) is a persistent problem that has not improved despite the now widespread use of parametric modeling in practice.

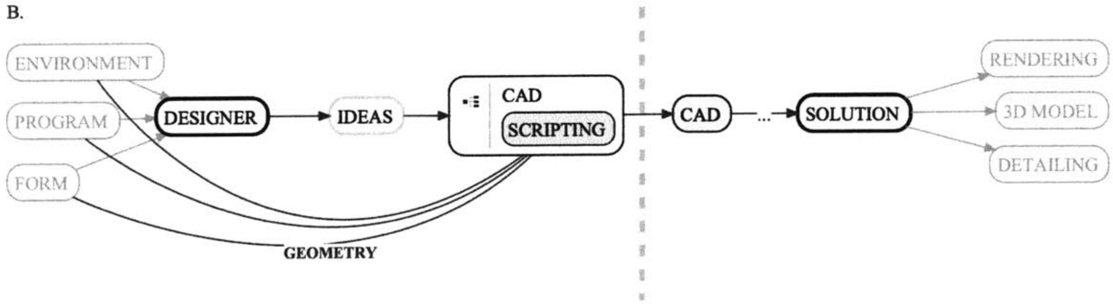


Figure 2-31: Contemporary design workflow models. [B] A mixed linear-feedback process where scripting in CAD provides a platform for geometric explorations when decisions are pretty much made.

Scripting is a less linear process where code embedded in CAD provides a platform for geometric explorations. A scripting language is a programming language that supports the writing of scripts, programs written for a special runtime environment that can interpret and automate the execution of tasks which could alternatively be executed one-by-one by a human operator. Most of these scripting techniques tend to automate CAD for tedious operations when design decisions are quite solid, but there are some that investigate generative design techniques to find form more in the spirit of applets for design (Figure 2-31). The difference between using scripting languages (in CAD) or high level languages (in applets) is that scripts are typically "quick and dirty" solutions that make the designer's life easier within a platform, somehow with the logic of macros, which are used to make a sequence of computing instructions available to the programmer as a single program statement, making the programming task less tedious and less error-prone. Also scripting languages have a relatively loose structure, so it would be difficult to use Java (a high level language) as a scripting language due to its stricter class and file system structure.

> *Generative Design is a morphogenetic process using algorithms structured as non-linear systems for endless unique and un-repeatable results performed by an idea-code, as in Nature.*
> *–Celestino Soddu[19], 1992*

The closest computer-aided design approach to wicked problems is that of generative design. It is a design method in which the output image, sound, architectural models, animation is generated by a set of rules or an Algorithm, normally by using a computer program. Most generative design is based on parametric modeling. Some generative schemes use genetic algorithms to create variations. Some use just random numbers. Generative design has been inspired by natural design processes, whereby designs are developed as genetic variations through mutation and crossovers. It is becoming more explored, largely due to new programming environments (Processing, vvvv, Quartz Composer, Open Frameworks) that have made it relatively easy, even for designers with little programming experience, to implement their ideas.

[19] Celestino Soddu is an architect and professor of Generative Design at Politecnico di Milano university in Italy. He is one of the pioneers of Generative Art and Design. His first generative software was designed in 1986 for creating 3D models of endless variations of typical Italian Medieval towns.

C.

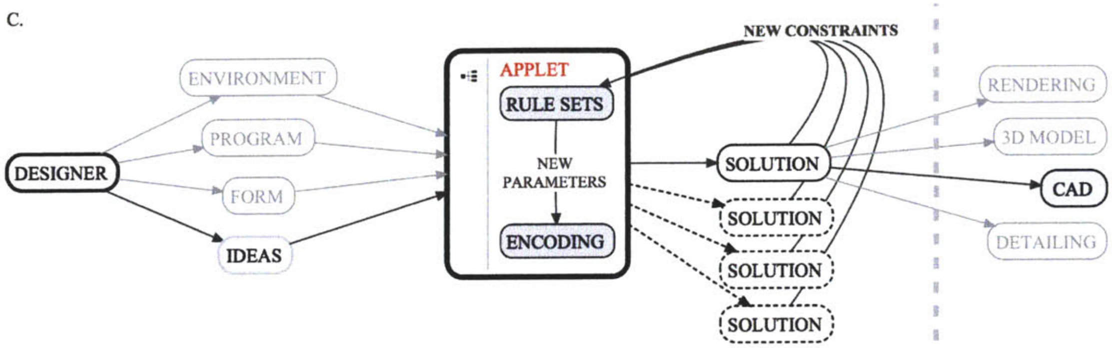


Figure 2-32: Contemporary design workflow models. [C] Applet-making is a problem-finding practice that allows for early intervention of constraints, pushes back representation and has an emphasis on creative aspects and design space exploration.

Applets (Figure 2-32) use object-oriented[20] high-level languages – enabling encapsulation, polymorphism ann inheritance[21] – which, by nature, are much able to scale up and embark in generative design explorations, than scripting ones[22]. Applet-making for design, as addressed in this thesis, is a problem-finding practice that encourages early intervention of constraints, delays the need for representation in the design workflow (using traditional CAD tools), and has an emphasis on creative aspects and design space exploration.

[20]Object-oriented programming (OOP) is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects are used to interact with one another to design applications and computer programs. C++ and Java are examples of object-oriented programming languages.

[21]Encapsulation refers to the creation of self-contained modules that bind processing functions to the data, and ensures good code modularity, which keeps routines separate and less prone to conflict with each other. Inheritance lets the structure and methods in one class pass down the hierarchy. The ability to reuse existing objects is considered a major advantage of object technology. Polymorphism lets programmers create procedures for objects whose exact type is not known until runtime.

[22]See section 3.3 for further detail and diagrams

Chapter 3

Proposal

Positioned between scripting agilities and software engineering practices, applet-making is envisioned as a way of legislating laws for a design exploration with results that we cannot imagine.

What is wrong with the tools we have?
The more and more we face architectural problems with conflicting constraints, with complex requirements involving innovative technologies, materials or uses. However, as explained in Chapter 2, traditional CAD software is generic in functionalities and constrained by a determined geometric kernel. Programming instead allows for rule-making to initiate open-ended processes. Applet-making, as proposed in this thesis, builds up on the emerging scripting cultures to create small stand-alone software applications that operate out of the limitations of specific kernel libraries.

Do designers today need to be software engineers?
No, applet-making is not a substitute for CAD, is something different, is intended to satisfy problem-finding in the design process. Applets are much smaller in functionalities that generic design environments. They are tailored for specific tasks. Software development (written with low level languages) is done by building full user interfaces, rendering engines, viewport systems, etc., instead applet-making (written with high level languages) takes advantage of frameworks (such as Cinder, OpenFrameworks or Processing) that provide the use, if desired, of minimal backbone functionality for the designer to build on and explore with aspects of design other than geometry.

3.1 Rule-Making Artisans

There are relevant precedents from diverse disciplines where creativity emerges from rule-making processes (Figure 3-1). When we observe their work we see rule-making for creative outcome, where the designer is the rule-maker, the master of the matter at hand, the one who understands and invents upon. While the computing device is what offers surprise by integrating the designer's thought, code, into a grater complex object, in the form of choreographies, structures, music pieces, patterns, installations etc.

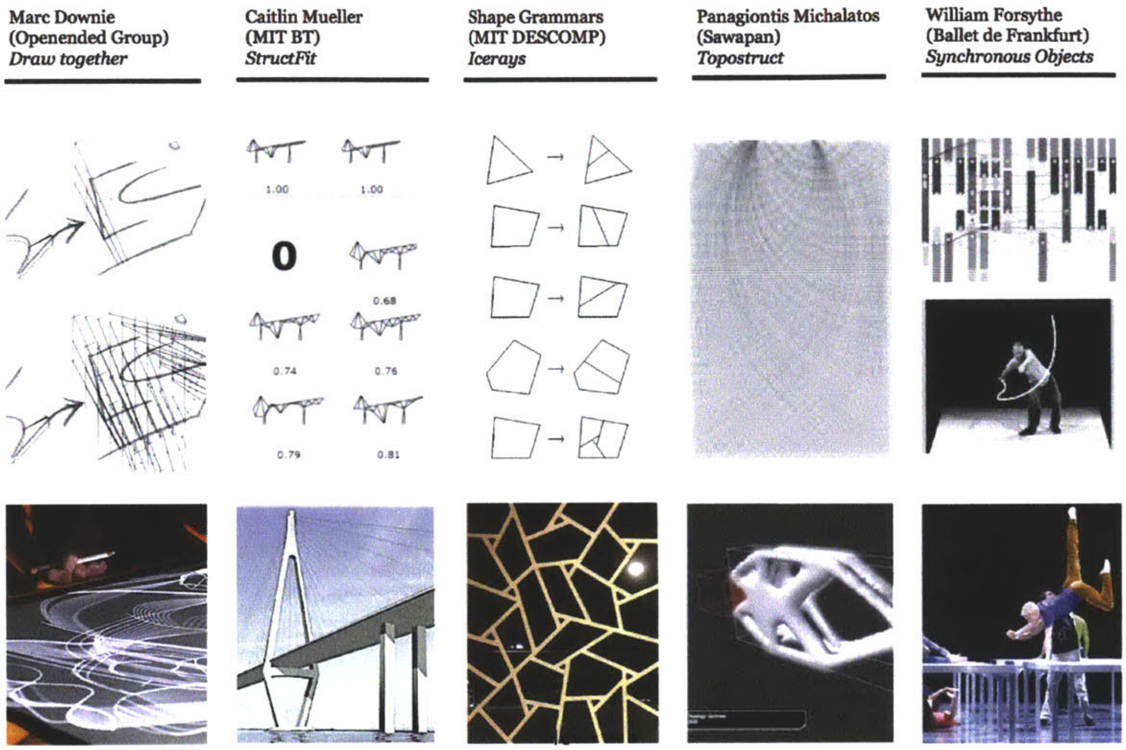


Figure 3-1: Five processes of making rules to design.

Marc Downie[1], co-founder of "OpenEndedGroup"[2], is a digital artist who builds algorithmic exploration tools to produce his artwork ranging from 3d projections, to digital music pieces, to innovative art installations (see Section 3.1.1 for further detail).

Caitlin Mueller[3] is an architect with engineering background who invents structural design tools that address digital brainstorming and conceptual design, rather than performance (see Section 3.1.4 for further detail).

George Stiny elaborated, in 1977, generative mechanisms for interpretation algorithms in aesthetic systems for a kind of seemingly irregular patterns called "ice-rays", a type of traditional Chinese lattice used in ornamental window grilles (see Section 3.1.2 for further detail).

William Forsythe[4] is an american dancer and dance company director. He is a classic ballet master who builds rule-based choreographies to design avant-garde pieces (see Section 3.1.3 for further detail).

[1] Marc Downie is an artist and artificial intelligence researcher with a PhD in MIT Media Lab.

[2] OpenEndedGroup comprises three digital artists Marc Downie, Shelley Eshkar, and Paul Kaiser whose pioneering approach to digital art frequently combines three signature elements: non-photorealistic 3D rendering; the incorporation of body movement by motion-capture and other means; and the autonomy of artworks directed or assisted by artificial intelligence (http://openendedgroup.com).

[3] Caitlin Mueller earned an undergraduate degree in Architecture from MIT and an MS in Structural Engineering from Stanford University. Her current research encompasses both aspects of her background, focusing on tools for the conceptual design of structures using interactive evolutionary algorithms. She is currently a Building Technology PhD student in MIT

[4] William Forsythe is known internationally for his work with the Ballet Frankfurt (19842004) and The Forsythe Company (2005present). He has produced and collaborated on numerous installation works that have been shown at the Louvre Museum, Venice Biennale, Artangel in London, Creative Time in New York, the Renaissance Society in Chicago, and other locations.

Panagiotis Michalatos[5] is an architect who builds explorative structural design tools to find from. He has developed a range of software applications for the intuitive and creative use of structural engineering methods in design (see Section 3.1.4 for further detail).

There is a common structure in the way this rule-makers think about design (Figure 3-2):

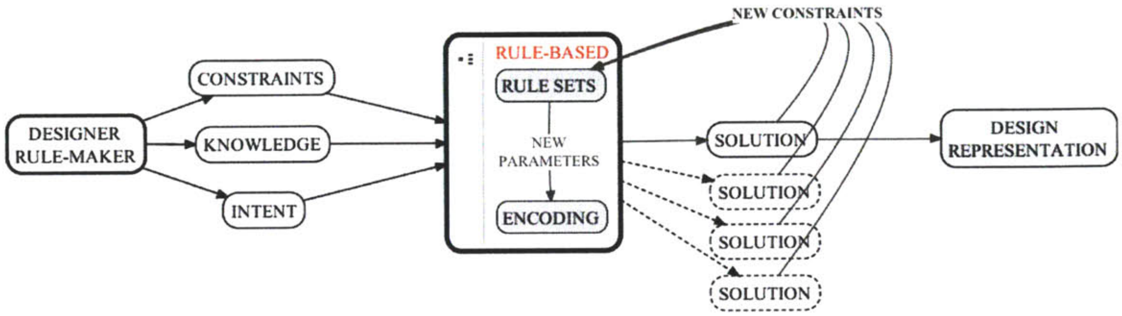


Figure 3-2: Rule-making process structure that the examples have in common.

- The designer performs an initial evaluation of the constraints that affect the system, in other words, defines the design space boundaries.
- Then evaluates which is the required knowledge that he needs to describe the constraints.
- And finally tries to convert his design intent into other rules that will add to the constraints set, these ones will not come from objective performance requirements but from subjective qualitative desires.
- Once that is clear – but not definitive, as it only initiates a conversation –, the rules, constraints and parameters are encoded into some kind of formal description, being programming languages, synchronous objects or mathematical models.
- The process of encoding these rules is a learning one of "designing the designing". Where design is research, and the designer comes to understand the phenomena that will help the exploration.
- The execution of the rule-based device – being code, transformation, dance, pattern etc. –, and its visualization, will inform about the new desired constraints to change it, or the new rules that will further describe it.
- The feedback process goes on an on with the designer as an active integrator in conversation with the computational device.
- Once one of the emergent solutions is picked as satisfactory – may not be the optima but this is a problem-finding not problem-solving process – the representation of the design will be assisted by tools such as CAD, or digital fabrication, or a dance company or an art installation.

[5]Panagiotis Michalatos is an architect with a MSc in applied IT from Chalmers Technical University in Sweden. He is currently working as a computational design researcher for the London based structural engineering firm AKT. While in AKT along with colleague Sawako Kaijima they provided consultancy and developed computational solutions for a range of high profile projects by architecture practices such as ZHA, Thomas Heatherwick, Fosters and Partners, Future Systems and others. Their work, in the development of computational design as a quasi-discipline in-between disciplines has been published and presented in international conferences.

3.1.1 Rules to design Installations

"Drawn Together" is an artwork installed at the Stubbins Studio Gallery (College of Architecture, Georgia Tech.) through February 13th to March 31st, 2012 (Figure 3-3). It has been designed in a way that it is actually an explorative drawing tool. The human cant direct the outcome, but the computational device can't either. The drawing that the two produce, in a kind of conversation, is an open-ended one, that involves different drawing media, music and a special pace, that will take the exploration to unexpected places.

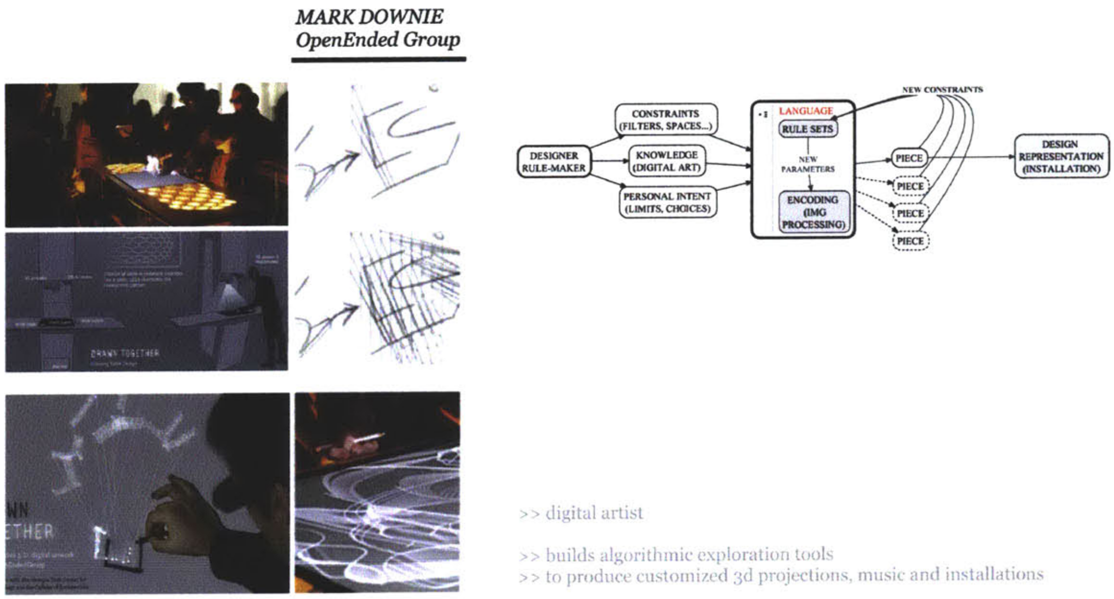


Figure 3-3: "Participants interact with a digital intelligence to create unforeseen and original drawings and music in 3d a drawing performance whose form is in equal parts physical and virtual" [OpenEndedGroup, 2012].

"Here's how it works. The physical manifestation of the installation, which echoes that of a drafting table, presents you with a variety of white drawing implements (chalk, gel-pen, pastel) with which you draw on a piece of black paper. While drawing you are also to put on headphones and a pair of 3d glasses. As you draw your first mark on the paper, you hear its trace amplified in the headphones. The curved body of the table is itself an electro-acoustic instrument, with a resonating chamber like that of a cello. When you withdraw your hand from the drawing, this signals the end of your turn. You then wait a moment for the computer to answer. It does so by projecting 3D lines that seem to draw themselves over, on, or under your paper. The virtual marks echo, extend, annotate, magnify, or complement the ones you've drawn physically, and as they draw in, the marks are accompanied by a music derived from the sound of your strokes. You continue in this fashion, taking turns with the computer, until you decide you're done which you signal to the computer by signing in the box it's projected on the lower right of the page. In staging this encounter between old and new forms of drafting, we swim against a strong current of our time. For Drawn Together takes us back to a slower, more deliberate pace its interaction gives none of the instantaneous feedback and reward of the computer games and programs we've grown accustomed to, but proceeds instead with all the appearance of care and reflection. So there is no prospect of mastery or of winning, but something like the opposite, for you can't direct the outcome (and nor can the computer). The "drawing" that the two of you produce as you take turns is an open-ended one that will take you and your thoughts to unexpected places." [OpenEndedGroup, 2012]

3.1.2 Rules to design Windows

Ice-rays are a kind of traditional Chinese lattice used in ornamental window grilles, they form irregular patterns. They are also observed in cracking ice on still water, where straight lines meet longer lines in innumerable ways. Stiny's exploration of the ice-rays is encoded in a formal mathematical device, and the beauty of it is that it does not constrain the possibilities of design, there is always something new to encounter while applying the rules (Figure 3-4).

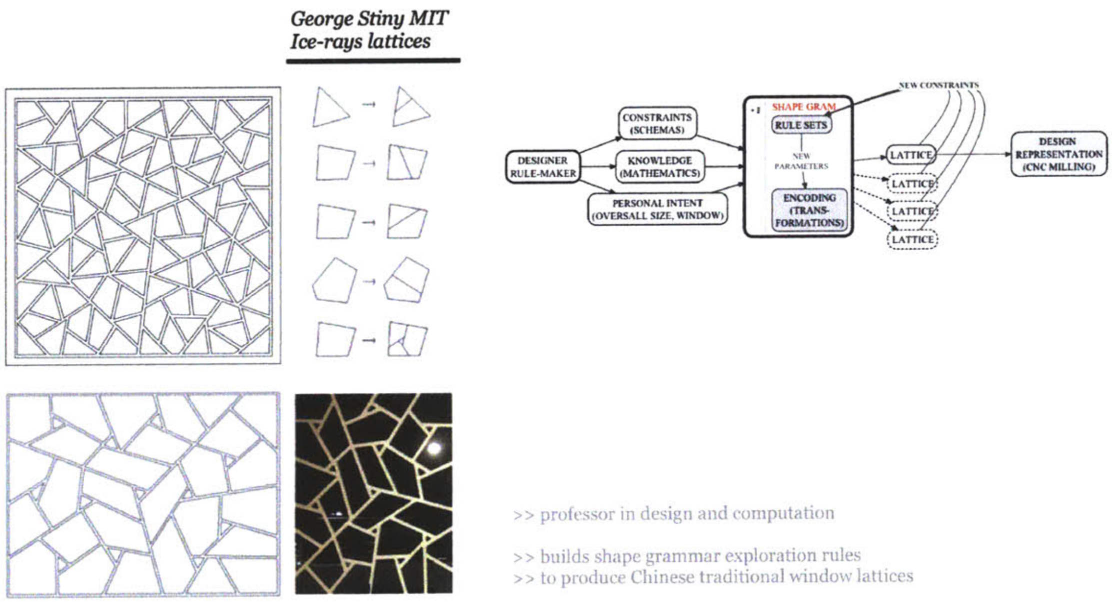


Figure 3-4: Ice-ray explorations created by shape grammar rules, and represented using CNC milling.

"Rules for ice-rays are easy to define in a shape grammar. They are in two equivalent schemas: $x \rightarrow \mathrm{div}(x)$ and $x \rightarrow x + x$, where x, x, and x are triangles, quadrilaterals, or pentagons, and div(x) divides x into x and x. Rules apply in alternative ways. Imagine a Chinese craftsman at a building site, with his tools and a trove of sticks. Shown a window opening, he starts an ice-ray. He selects a stick of the right size and inserts it between two sides of the rectangular frame to form two quadrilaterals. He continues his work by dividing one of these areas into a triangle and a pentagon. Then he divides the triangle into a triangle and a quadrilateral, and the pentagon into a quadrilateral and a pentagon. He goes on connecting sides of polygons to make others of the same kind. Everything is stable in this process, if he keeps to the rules. It is striking how rules apply recursively, but also notice something new. Calculating in a shape grammar is visual. Rules apply directly to ice-rays. There are no hidden representations that limit what you can do what there is, you can see, and what you see is there. Divisions in ice-rays may vary some require multi-axial figures and motifs. Just put them in rules: draw what you see before you divide, and then draw what you want. Or let the schema $x \rightarrow \mathrm{div}(x)$ include your rules, so that polygons are divided into two areas or more. There is no end to the ice-rays you can get from the schema $x \rightarrow \mathrm{div}(x)$: both known ones, and ones that are new. Go on and try the schema make an ice-ray of your own!" [Stiny, 2008]

3.1.3 Rules to design Choreographies

When William Forsythe observes his dance company perform, he experiences the emergence of a multiplicity of rules (objects) (Figure 3-5). The the rules are body movements for his dancers to integrate, react to and dance with. It is not until all the rules are at play that the choreographer discovers the piece.
Again, the rule-making here comes from a deep understanding of the rules of classic ballet (the knowledge part in Figure 3-2), and the need to twist them and maybe break them to adapt them to the designer's needs. Here the computer is the dance ensemble, producing a new piece at every run, computing all the relationships and reactions at every move.

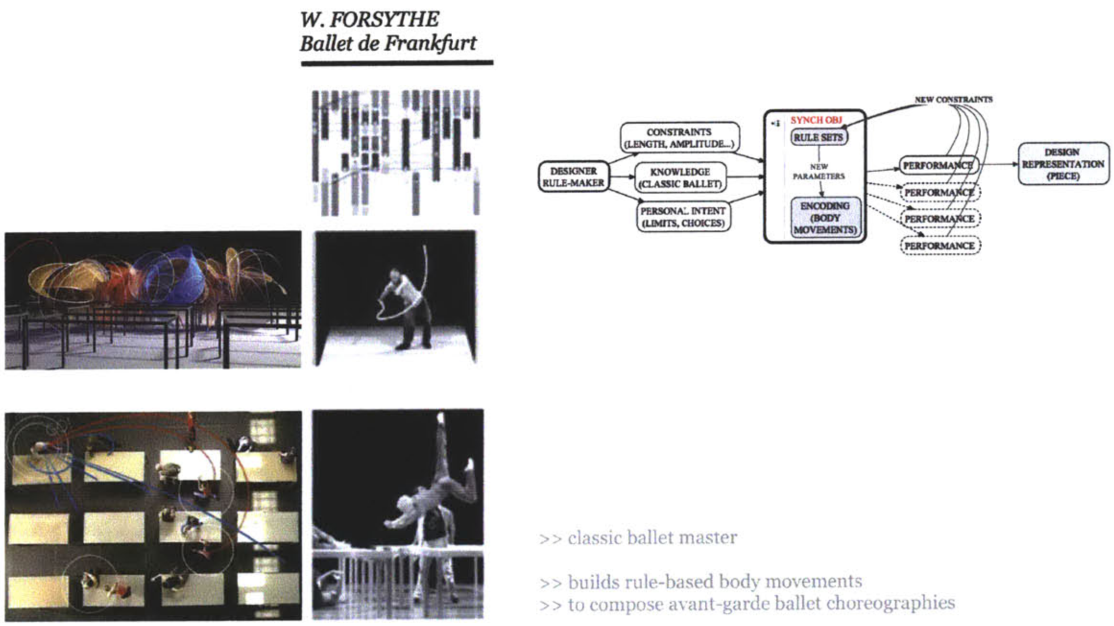


Figure 3-5: William Forsythe - Rule-Based emergent choreographies from body movement encoding.

"Synchronous Objects"[6] is an initiative that investigates the interlocking systems of organization in the choreography of Forsythe's piece "One flat thing reproduced" (2000). The data in the systems is collected and transformed into a series of synchronous objects to explore the choreographic structures, reveal their patterns and re-imagine what else they might look like.
"Choreography elicits action upon action: an environment of grammatical rule governed by exception, the contradiction of absolute proof visibly in agreement with the demonstration of its own failure. Choreography's manifold incarnations are a perfect ecology of idea-logics; they do not insist on a single path to form-of-thought and persist in the hope of being without enduring" (...) A choreographic object is not a substitute for the body, but rather an alternative site for the understanding of potential instigation and organization of action to reside. Ideally, choreographic ideas in this form would draw an attentive, diverse readership that would eventually understand and, hopefully, champion the innumerable manifestations, old and new, of choreographic thinking. [Forsythe, 2009].

[6]http://synchronousobjects.osu.edu

3.1.4 Rules to design Structures

Michalatos and Mueller build their structural exploration tools, in a similar process to that of applet-making. Their computational methods are those from engineering optimization but also incorporate an explorative component in the sense that their tools are fully interactive. At every human intervention the computer responds with a structurally efficient form, but it never distorts the designer problem set up: meaning that boundary conditions, loads, and support placement are in the hands of the designer, while the solver will compute structural performance accordingly. Decision making is here assisted by a constant visual representation of the structure shape.

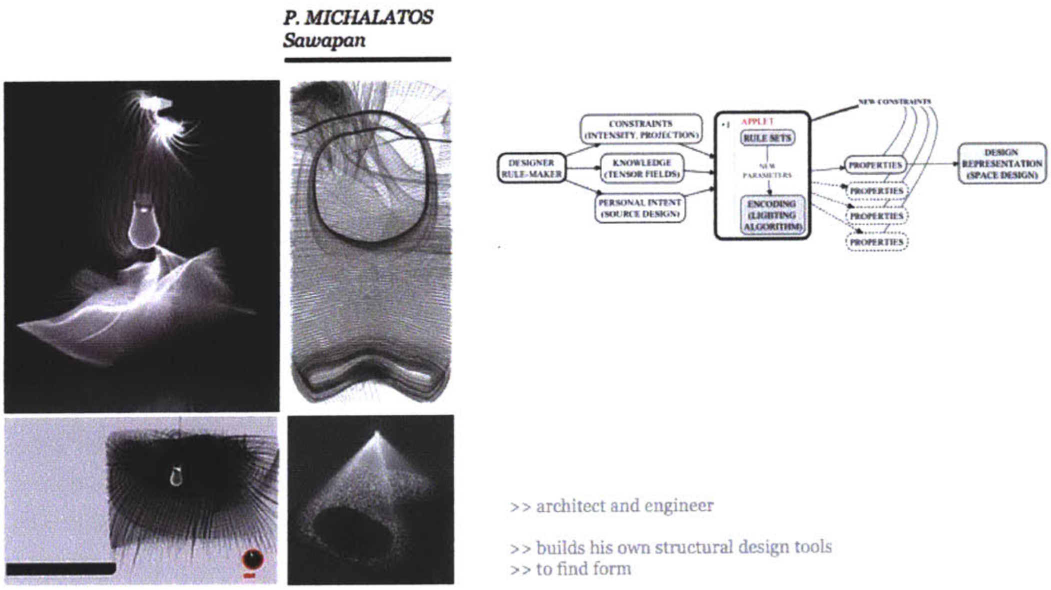


Figure 3-6: Panagiotis Michalatos - Applet-Making to design spaces through lighting constraints.

One of Michalatos's applets is called "Tensor Shades" where he explores light shades generated by the interrogation of structural information (stress tensor field) and desired lighting information (inverse illumination field) (Figure 3-6). Its inspiration comes from a frequent professional observation of a disjunction in the relationships between design intentions, geometry, and structural considerations. This is partly a result of the division of disciplines within the design and construction industry, and reinforcement of such divisions by the employed methods and software tools. Such tools allow designers to create more complex forms than previous technologies could support, yet they treat the design space as homogeneous and isotropic Cartesian space. As a consequence, operations on a designed object tend to disregard its intrinsic properties and behavior, making integration and negotiation of concerns difficult. To overcome this problem, the applet focuses on the construction and interrogation of tensor fields that endow the design space with varying properties. The ambient space properties will affect and enrich the embedded objects [Michalatos and Kaijima, 2008].

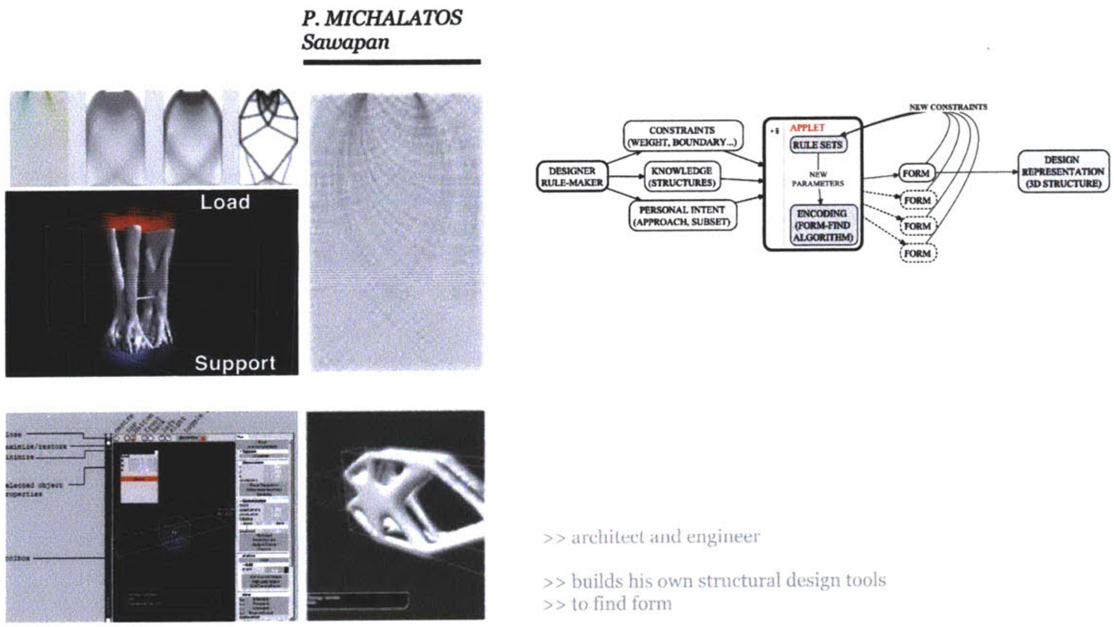


Figure 3-7: Panagiotis Michalatos - Applet-Making to find form through structural design constraints.

Another of Michalatos's most explorative applets is called "Graphemes" (Figure 3-7), it is "an experimental design program based on graph theory and a dynamic spring-force model. It has two main functions. First the design of 2 and 3 dimensional structures through the use of topological operations where the emphasis of the interface design is on connectivity of nodes. The real time spring model simulation will regularize the structure automatically. The second function of the program is to try and develop the structural intuition of the user by providing fast structural analysis of the design system. Hence one can observe in near real time how changes in connectivity in a space frame affect its structural behavior" [Michalatos, 2009].

The difference between these applets and applet-making as understood in this thesis, is that here the designer is also tool-maker for other designers to be tool-users. The approach of applet-making is that the designer makes her own tools that are adapted to every design project and that are never as general as to be used by a broad community targeting different design problems. Back to Weizenbaum's quote "one programs not because one understands, but in order to come to understand" [Weizenbaum, 1976], programming is an act of design, and by using tools, or by using other designer's applets the exploration we will be constrained by their assumptions and goals.

What the chosen precedents have in common is their primal intent to create rules for small specific systems. Their rule-making processes help them invent and produce something that was not possible to predict in advance. Personal rule-making is a way to operate temporarily in design, then representation, fabrication and installation process will take care of the physical construction of the exploration, as traditional CAD does in architectural design.

3.2 New Design Companions

Digital design is now fully assimilated into design practice, and we are moving from an era of being aspiring expert users to one of being adept digital toolmakers. –Mark Burry, 2011

3.2.1 Design the Designing

Peter von Buelow, Panagiotis Michalatos, Caitlin Mueller, and many other researchers, are investigating how to develop software tools that address the "creative phase" of design projects, there where the first decisions still need to be made. Buelow's proposal tackles some issues of the traditional genetic algorithms optimization techniques. He argues that they typically seek one best solution in the design space, and are better suited for later design phases. Instead he argues that his design exploration tool tries to expose a range of considerably good solutions, while aiding ideation and creativity in early design phases. Just by seeing the results of the computation, he argues, we are exposed to parts of the problem that we are not aware of, that we have not initially thought about. He also claims that it is not about looking at random forms but at constrained ones that still belong to a broad design space [von Buelow, 2007].
Buelow's tool tackles important questions about the design space exploration. While conceiving new designs, the exploration of constrained but broad possibilities enables the discovery of different ways to see the problem. What if a source of major inspiration is hidden in a solution that the tool has automatically discarded because of a not ideal performance criteria? This is a crucial difference between engineering convergent problem-solving and architectural problem-finding. The best solution for a building may not be the one that performs better in a range of quantifiable criteria. The only problem with Buelow's approach is that it is building the rules for the designer to use. Even if it is given a bigger set of design space solutions, it is not the designer who decides and describes the rules, who has control over the backbones of the evaluation criteria.

It is true that architects cannot be also computer scientists and engineers. However, can we have a say in the tools we use? Architectural design development is always specific to a project, so learning how software tools operate will tell us if they are suited for unique problems. In a paper from the last, 2012, eCAADe Conference entitled "Collaborative Digital Design: When the architect meets the software engineer", the idea of architects becoming expert programmers is discussed. The authors argue for a close collaboration between the architect and the software engineer in order to overcome current frustrations encountered in computer assisted design tools. One of the reasons they provide is that "at this moment, architects are not capable of overcoming some programming problems by themselves because they do not possess the necessary knowledge to distinguish between problems that result from legacy systems, language implementation details, and inadequate programming approaches, from those that are intrinsically complex from the computational point of view" [Santos et al., 2012]. Therefore software development becomes messy and ineffective. They claim that software engineers scan propose solutions to overcome those low level programming problems that are fundamental, but secondary to design, so that the designer is able to focus on the design the designing task. Of course, this dialogue between computer scientist and architect can only occur if one understands part of the other's task.

The process of applet-making does not require architects to be software engineers. It only requires them to go one step beyond contemporary practices of scripting and visual programming in CAD environments. It does not require proficiency in low level programming but on high level languages and the understanding of their libraries. "Learning programming requires learning a programming language, and it is only reasonable that architecture curricula adopt languages that can be directly applicable in the production of digital design. As a result, most curricula teach languages that are provided by CAD applications, for example, AutoLisp, Rhino-Script, and Grasshopper. Even though these languages allow a smooth transition from theory to practice, they have two major problems, namely, (1) most of these languages lack fundamental pedagogical qualities, and (2) professionals become locked-in to specific CAD languages and tools because they find it difficult to learn and adapt to new systems" [Santos et al., 2012].

In order to avoid the problems explained above, learning high level programming languages (like Java, Csharp or VB) provides:

- a better symbiosis between design thinking structures and programming structures
- the computational thinking desired to be able to switch between object oriented languages
- the freedom from the geometric kernels that prescribe scripting in CAD
- the ability to push the need for CAD representation systems to the latest moment of the design process

Back to Weizenbaum's quote, "one programs not because one understands, but in order to come to understand. Programming is an act of design. To write a program is to legislate the laws for a world one first has to create in imagination". The fact that by applet-making the designer needs to truly understand some of the fundaments of what he is designing with, makes him better equipped for design exploration and research-based approaches to the complex problems that challenge architecture today. For example there is research involved in the implementation of the optimization method chosen for a rooftop form-finding algorithm, or for the simulation of a cellular automata for landscape planning, or in oder to visualize a complex data set that will reveal the wealth of a neighborhood for further urban planning, or to find the geometric language that fits a fabrication method and a specific material. In a more concrete example we can be interested in designs informed by light. We can investigate that by building an applet that will simulate a light source and trace its shadows. It may obey the laws of a sun or a lamp for a determined location. But of course to build it we will need to master the rules of light. It is not the computation technique or skill that matters here, but the understanding of the phenomena demonstrated by our ability to translate it into a description, into rules, into code. Then we let our computer companion trace the rays, represent the outcome of our rule-making and surprise us, deliver propositions that we integrate or revise or re-run again with different rules and parameters or constraints.

"The understanding of programming is one route to a more thorough understanding and exploitation of the computers potential. Programmatically, a two-fold approach to finding form is necessary to explore design possibilities. First, one needs to frame the problem, and then one may explore the domain of possibilities that the frame establishes" [Burry et al., 2000]. If something needs to be written in the form of code, it needs to be understood first.

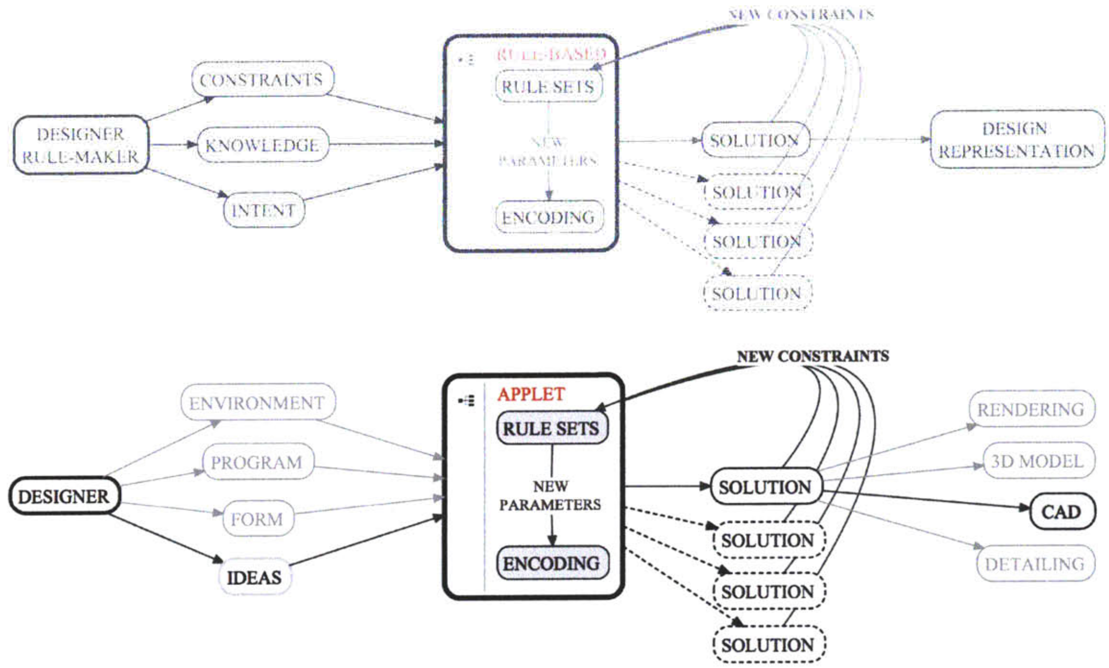


Figure 3-8: Applet-making is a specific case of rule-making for design.

As we have seen in Section 1.1, rule-making involves a set of *constraints*, a body of *knowledge* that we are willing to learn and a design *intent*. In the specific case of applet-making for design, these 3 premises are mapped to an *environment* where the solution is going to live (city, building, object, body part, etc), a *program* which defines the function of the design (urban strategy, rooftop, actuator casing, body armor, etc), a phenomena that we need to understand that will determine *form* (being data inspired, load inspired, fabrication inspired or biologically inspired), and finally a set of *ideas* or desires from the designer's imaginary or qualitative considerations (Figure 3-8).

The applet-making case studies to follow, span four scales of resolution: city, building, object and body. They also belong to different types of applets referring to their main computational goal: visualization, form-finding, optimization, and translation. They have been chosen, not to provide a comprehensive survey of the emerging culture or its possibilities, but to demonstrate different scales of operation and design problems of different nature that require tools having the following characteristics:

- **Results challenge human Prediction**: it is not possible to imagine a form, hence not possible to build it in any CAD modeling tool.
- **Problem spans Conflicting Constraints**: there exist constraints that belong to different fields or that seem to contradict one another.
- **Need to satisfy Complex Requirements**: requirements spanning comfort, kinetics, structural performance or fabrication processes.
- **An Open-Ended solution space**: the solution does not have to score an optima but a satisfactory performance in relation to the constraints.
- **Negotiation and Human Integration required**: subjective and qualitative aspects may change the rules of the system.
- **Exploration described by an Initial Set of Rules**: description of seed rules will engage the human and computer in a conversation.
- **Understanding one or more Specific Phenomena**: the complex requirements demand that the designer embarks on a learning adventure.
- **Small set of Tailored Functionalities**: the applet will target the least possible functionalities to serve a specific task.
- **Rules shift and emerge by Conversation**: the system will accommodate adjustments, changes and new decisions in every iteration.

3.2.2 Companion for city Planning

In this case, the designers are in need of a tool to assist with decision making in strategic urban planning. The need for the applet presented here comes from a lack of diagnose of specific but rare urban indicator. The urban indicator at stake is the amount of artists in need of creative space and residence in every neighborhood of a two million people city. With no capabilities to conduct a city-wide survey, a generative rule-based process is undertaken.

The generated tool manages three different types of data. First general space requirements for a creative space according to specific arts discipline, second the existing urban indicators per neighborhood, third a set of space appropriation strategies defined by the designer's background in the seek of opportunity spaces.

The space requirements and urban indicators are quantitative data that can be contrasted, but it is normally presented the form of spreadsheets. However, the designers need to "see" its interplay in order to make informed decisions that will shape the urban tissue and favor certain neighborhoods against others. The tool makes that seeing possible by applying filters and different ways to visualize the diagnostic, so that the exploration of the data sets can reveal new ways to look at the city. Also, back to the fact that programming is a way of coming to understand, by massaging the data sets, the designers get a complete understanding of the problem and can better apply formal models for its visualization [Mogas-Soldevila, 2009].

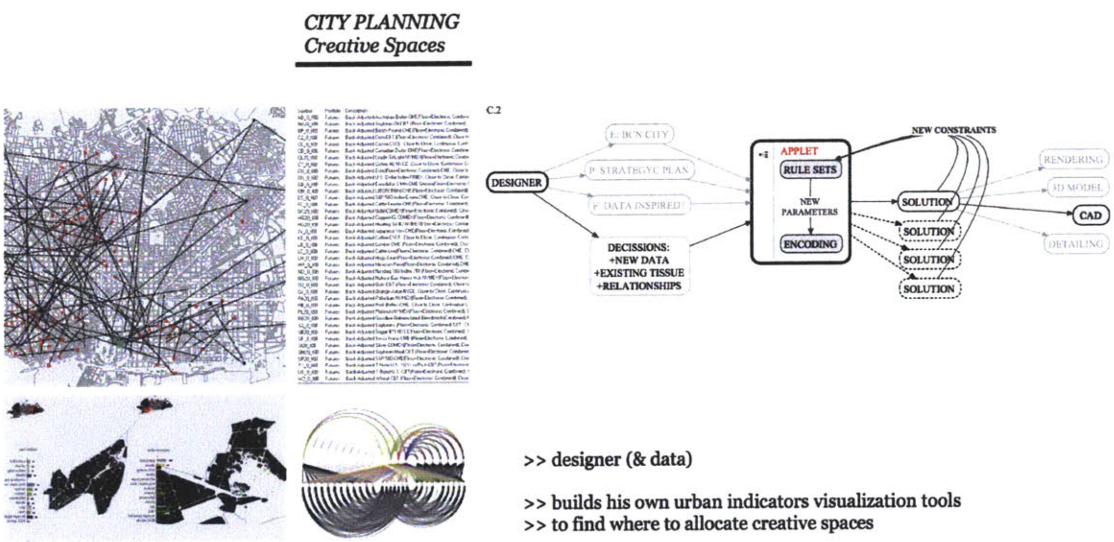


Figure 3-9: Visualization Applet. Open up city-scale strategies through Data managing (development by the author in collaboration with Jorge Duro-Royo[7] [Mogas-Soldevila, 2009].

- **Environment**: a city of two million people that needs to allocate new creative spaces for artists.
- **Program**: consolidate a strategic plan that places the spaces there where it is more convenient for the community.
- **Form**: the positioning of the spaces presents a problem, there is no data that identifies current locations of artists in need of studio in the city fabric, or that predicts where it would be more convenient for them to move.
- **Decisions**: new data needs to be collected and existing urban indicators will help with that, after the processing of the data, a visualization tool that diagnoses every neighborhood will provide the city planners with a platform where to make informed decisions.

3.2.3 Companion for roof Shaping

In this case, the designers are in need of a tool to assist with decision making in roof shaping. The need for the applet presented here comes from the impossibility to imagine an outcome of different conflicting constraints affecting form. The conflicting constraints at stake are load distribution of the new roof into the existing building, provide light and ventilation under the new structure, and the desire to make the rooftop walkable and a rain water collector by shape. With no capabilities to model or sketch the shape responding to the requirements, a generative rule-based process was undertaken.

The generated tool manages a structural form-finding model that will calculate the most lightweight shell for the decided support points, and allow for the interactive opening of ventilation patios.

The structural calculation is quantitative data that can be easily represented. However, the designer needs to "see" its interplay with the rest of constraints, in order to make qualitative decisions that will affect the walkability of the roof and the design of the space below. The tool makes that seeing possible by representing the structural results over the existing building, so it becomes a tool to negotiate with. Also, back to the fact that programming is a way of coming to understand, by coding the structural form-finding algorithm, the designer gets a complete understanding of the problem and can better decide how the qualitative constraints affect the performance of the roof [Mogas-Soldevila, 2009].

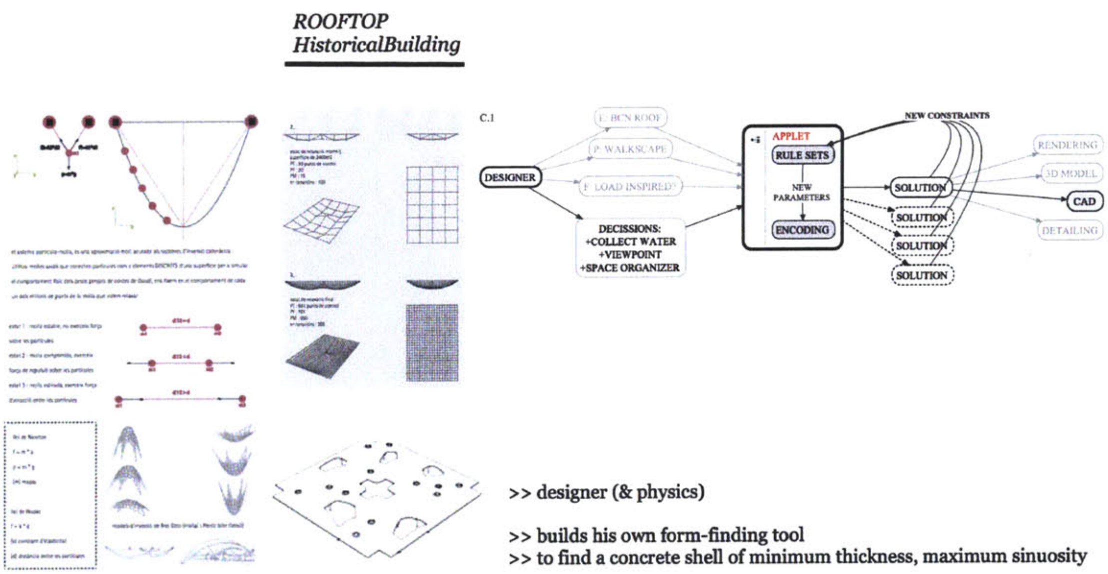


Figure 3-10: Simulation Applet. Explore the lightweight shapes for a roof over an historical building (development by the author in collaboration with Jorge Duro-Royo) [Mogas-Soldevila, 2009].

- **Environment**: the top of an obsolete historical water tank.
- **Program**: the initiative to create a roof over the building to allocate new functions where the water used to be.
- **Form**: the most pressing constraint is that of load, the existing structure needs to receive the new loads in a certain way, but other functional constraints apply, such as the need for light and ventilation under the new roof and the possibility for it to be walkable.
- **Decisions**: the designers also decided that the roof should collect rain water, provide a landscape view point and its structure should help organize the space below.

3.2.4 Companion for system Translation

In this case, the designers are in need of a tool to assist with the translation of a prehistoric fish exoskeleton system to an armor for a human chest. The need for the applet presented here comes from the impossibility to describe the system rules on a new host. The conflicting constraints at stake are the need to protect the body and also provide for certain reneges of motion. With no capabilities to model or sketch the shape responding to the requirements, a generative rule-based process was undertaken.

The generated tool abstracts the fish scale system into three levels of understanding and definition: local, regional and global rules. The rules are geometric relations that make possible that every scale properly connects and overlaps with its neighbors, complying with protective and kinetic constraints.

The data structure of the component system is quantitative data that can be represented by a set of algorithms. However, the designer needs to "see" its interplay over the chest of a human host, in order to make qualitative but performance-informed decisions that will affect the final design. The tool makes that seeing possible by representing the results according to the designer tracing of the underling surface, it becomes a negotiation tool. Also, back to the fact that programming is a way of coming to understand, by coding the algorithms for local, regional and global rule systems, the designer gets a better understanding of the biological organism and the way nature negotiates between protection and kinetics by gradating geometry.

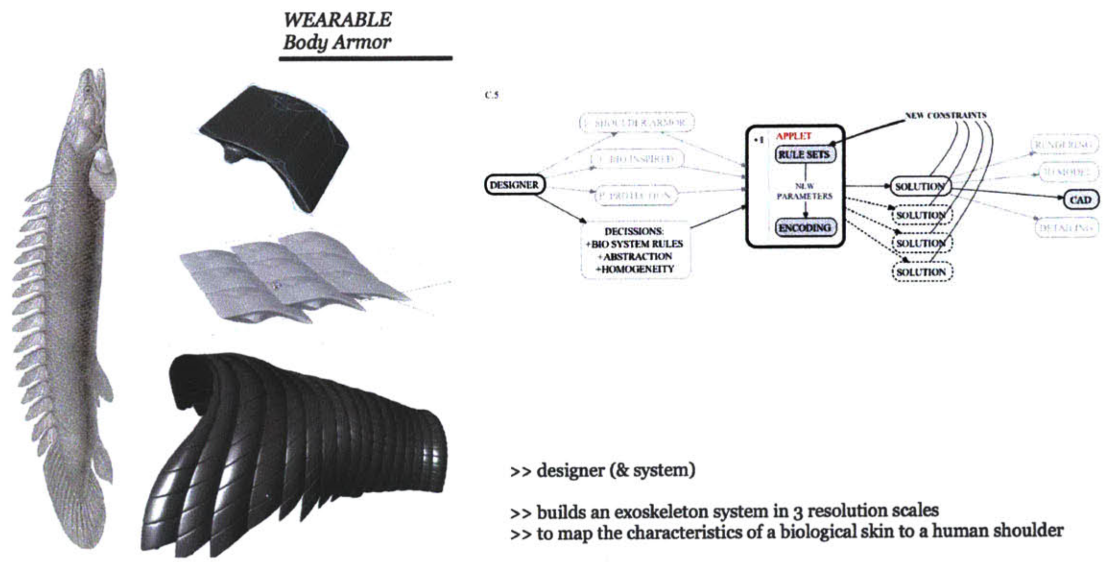


Figure 3-11: Translation Applet. Wearable protective system definition for application over different hosts (development by Jorge Duro-Royo with assistance of the author).

- **Environment**: the human body of a solider.
- **Program**: protect the person while allowing for the normal ranges of motion of a solider in combat situations.
- **Form**: the shape of the armor will be determined by a biological system, the skin of a prehistorical fish made of ceramic scales that protect its internal organs and allow, as well, for extreme swimming flexibility.
- **Decisions**: the designers decided to explore the organism skin geometric rules in order to abstract them and translate the logic to a new host while optimizing the system for maximum component homogeneity.

3.2.5 Companion for fabrication Fit

In this case, the designer is in need of a tool to assist with decision making in a component shaping. The need for the applet presented here comes from the impossibility to describe the geometry of a carbon-fiber composite actuator casing that would fit its fabrication process constraints. The conflicting constraints at stake are the need to insulate an actuator, refine a geometry for composite molding, and the desire to design an aesthetically compelling casing. With no capabilities to model or sketch the shape responding to the requirements, a generative rule-based process was undertaken.

The generated tool manages a new mesh representation technique that provides the designer with further control of the subdivision at every edge of the geometry. So that, through an interactive process of adapting the new shape to the existing actuator, the designer can both comply with molding fabrication constraints and aesthetic intentions.

The mesh subdivision calculation is quantitative data that can be represented by the algorithm. However, the designer needs to "see" its interplay with the rest of constraints, in order to make qualitative decisions that will affect the perception of an organically-shaped casing. The tool makes that seeing possible by representing the meshing results according to the designer manipulation of a basic set of control points, it becomes a negotiation tool. Also, back to the fact that programming is a way of coming to understand, by coding the mesh data algorithm, the designer gets a complete understanding of the problem and can better decide how CAD tools respond to similar problems.

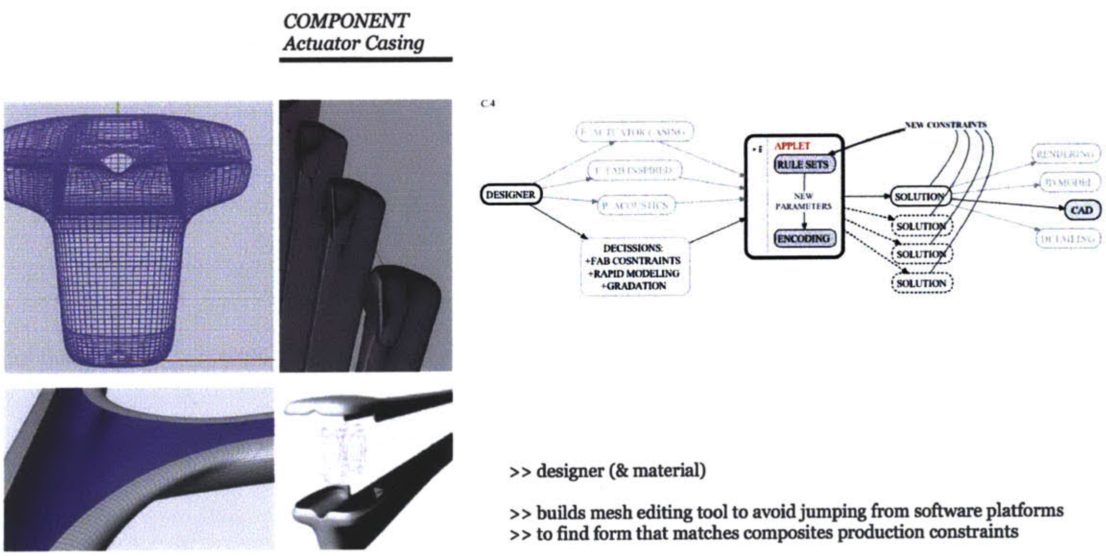


Figure 3-12: Optimization Applet. Tailor form for specific Material and Fabrication constraints (development by Jorge Duro-Royo).

- **Environment**: the casing for an electrical actuator in a dynamic skin project.
- **Program**: the main goal of the casing is to protect the environment from the acoustical nose of the actuator.
- **Form**: the form of the casing is determined by the material that will conform it, carbon fiber composite.
- **Decisions**: the designers decided to build a custom mesh representation in order to fit the geometry to the fabrication process, so they could control every aesthetically aspect in the final product.

3.3 Scope and Detail

Applet-making is a specific case of rule-making for design, a case of world-making in the computer. Different programming languages provide different types of abstraction depending on the intended applications. Applets, in the context of this thesis, are written in modern high level[8] compiled[9] languages implementing OOP paradigm[10], which provide; in one hand, better adaptation to the designer thought process through the use of natural language, and in the other hand a basic library of functionalities (such as user interface helpers, mathematical solvers of different sorts, geometric primitives, vector operators etc.) that can be the backbone of the applet.
The kind of worlds that applets build can be described by:

- **Elements**: such as shapes or objects or entities, normally defined by a class[11].
- **Relations**: or rules that apply to the elements, normally described as class methods[12] or functions[13]
- **Constraints**: to restrict the system from achieving its potential goal by assigning decision thresholds.
- **Simulation**: the abstract representation of the system of elements and relations, a technical universe.

By world-making we try to build rules to describe phenomena. These rules are spatial relations between elements such as shapes or objects or entities. The relations will be nuanced by the values of the constraints that the designer decides. The system elements are normally represented as geometries or pixels in the screen, but behave in higher complexity and have characteristics – data members and methods – other than geometrical. In the computer, the world is represented in a simulation that Habraken[14] describes as "technical universe" [Habraken, 1987]. During the simulation process, the need for new rules and new perception of design may emerge.

The difference between applet-making and scripting or visual programming within CAD, mainly lays in the fact that, with applets, the designer can fully deploy his natural rule-making thought process. In OOP, world-making is enabled by global variables definition that affect the whole system, local variables described in abstract classes, subclasses that extend others, functions that affect subclasses, functions that affect the world, interface methods that can be invoked to class instances without being part of the class, etc. The richness of world-making is enhanced. Also by manipulating programming logics, the designer can better understand the interplay of the rules and the element description of his design system (Figure 3-13).

[8]"High-level language" refers to the higher level of abstraction from machine language. Rather than dealing with registers, memory addresses and call stacks, high-level languages deal with variables, arrays, objects, complex arithmetic or boolean expressions, subroutines and functions, loops, threads, locks, and other abstract computer science concepts, with a focus on usability over optimal program efficiency.

[9]A compiled language is a programming language whose implementations are typically compilers (translators which generate machine code from source code), and not interpreters (step-by-step executors of source code, where no pre-runtime translation takes place).

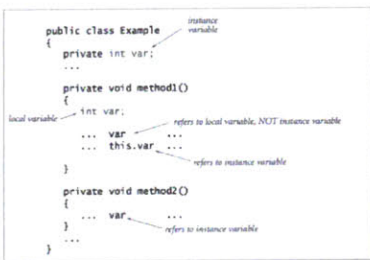
[10]Object-oriented programming (OOP) is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs. C++ and Java are examples of object oriented programming languages.

[11]In OOP, a class is a construct that is used to create instances of itself referred to as class instances, class objects, instance objects or simply objects. A class defines constituent members which enable its instances to have state and behavior. Data field members (member variables or instance variables) enable a class instance to maintain state. Other kinds of members, especially methods, enable the behavior of class instances.

[12]In OOP, a method is a subroutine (or procedure) associated with a class. Methods define the behavior to be exhibited by instances of the associated class at program run time.

[13]In computer programming, a function or subroutine is a sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed. Subprograms may be defined within programs, or separately in libraries that can be used by multiple programs.

[14]N. John Habraken is a Dutch architect, educator, and theorist. His major contributions are in the field of mass housing integrating users into the design process, and on the field of computer aids to design and concept games.



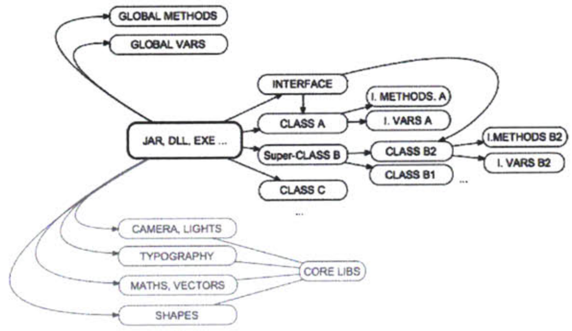


Figure 3-13: Using the object-oriented programming paradigm with high level languages, applets enable richer environments to think with and see differently. Descriptions can be formally defined by the user and so, their objects, methods and constraints changed to adapt to shifts in the explorations.

Another difference between scripting and applet-making is, as we have seen, the fact that the libraries (standard to the language or created by third-party designers) available to the rule-maker are vast and from many different fields other than CAD's geometric operations – such as sound, hardware interface, animation, video, mathematics, physics, data management, import and export protocols, typography, simulation, graphic interface etc.

There asr many different kinds of applets, as much as designers and design exploration needs, from visualization, to optimization, form-finding, generative, evolutionary etc., all of them with a basic set of Rules, Elements and Constraints.

If we look closer the the applet anatomy and to the applet-making process we can observe a common logic depicting an exploration intention translated into code and simulated. After that simulation, rules, elements and constraints can be changed by the designer to allow for new ways of seeing. What is true though is that there is no straight forward correspondence between design rules and programming functions, or between design elements and language classes, or between design constraints and programming parameters, as there is no "code" translation of human thought (Figure 3-14). What we can observe here is the designer integrating and negotiating with the computer in a back and forth process of discovery through intent, encoding, seeing and changing.

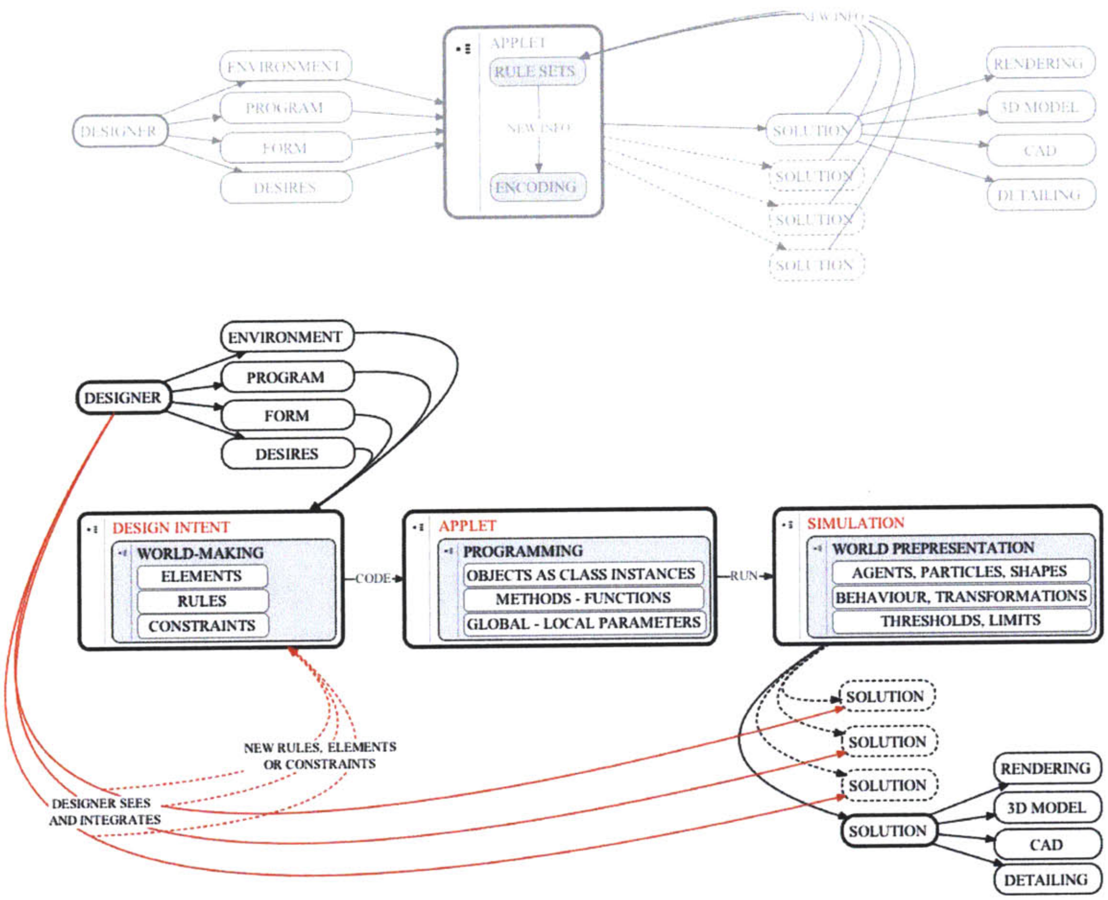


Figure 3-14: The complexity of world-making for design may be encoded into a design applet, but it will never fully substitute the designer's integration and negotiation while observing his design world simulation.

3.3.1 Self-Made and Ready-Made Computation

Some technical references in this thesis require further detail on levels of computational design abstraction and the meaning of Self-Made Computation. First of all, there are many ways to engage in a creative process with the computer; from using our body language as input, to building from scratch the full computer system. I argue that, the more rule-making – the more rules we describe for the computer to elaborate on –, the more design becomes inductive reasoning[15], and the closer we get to participate in a creative process with the computer. In Figure 3-15 below, I position the scope of this thesis in a classification of these levels of engagement.

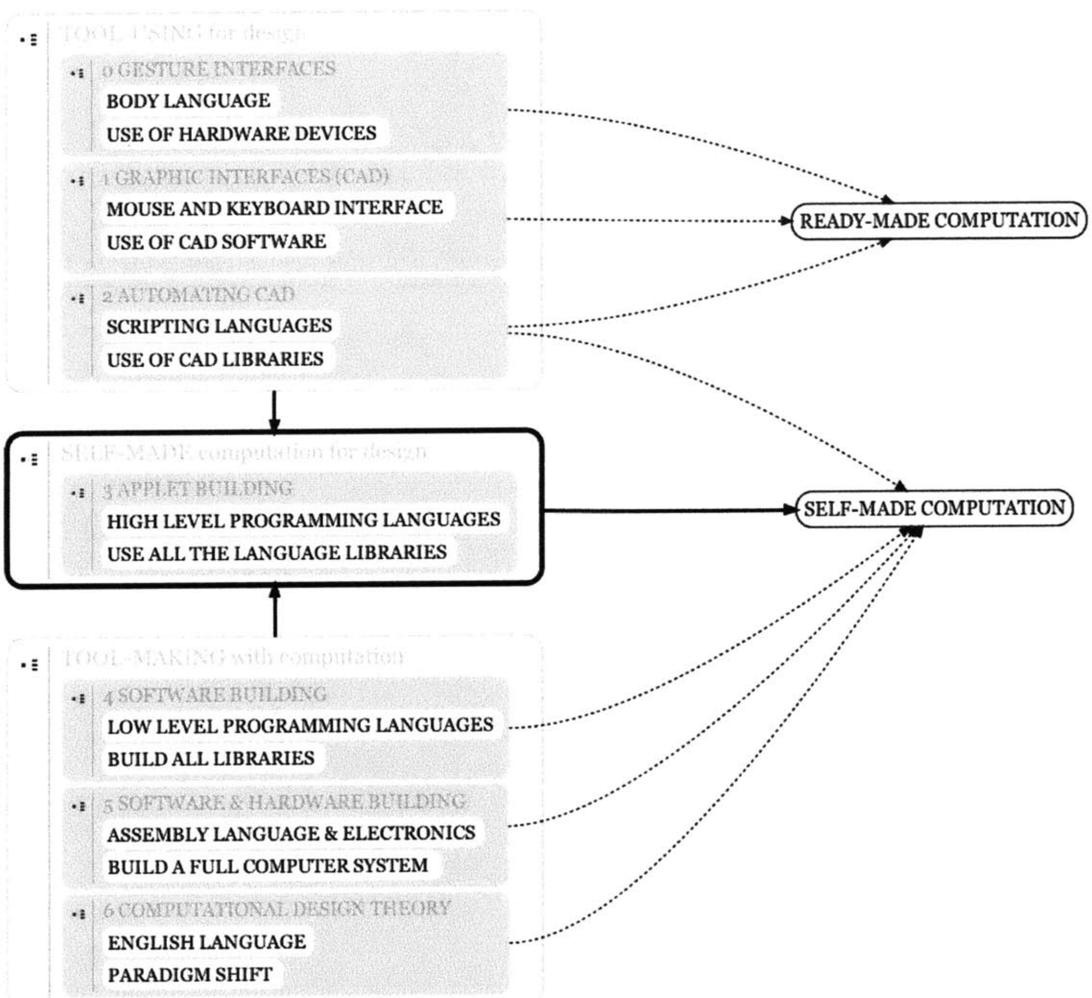


Figure 3-15: Abstracted, drawn and expanded from Cardoso, D., 'Hello (3D) World: An Introduction to Geometry Automation using RhinoScript' in conversation with the author.

Applets for design are dedicated to perform small specific tasks such as word processors, simple computer games, internet browsers etc. They are software applications as well as CAD software but target a much narrower set of functionalities (Figure 3-16).

[15]Inductive reasoning, also known as induction or informally "bottom-up" logic, is a kind of reasoning that constructs or evaluates general propositions that are derived from specific examples. Inductive reasoning contrasts with deductive reasoning, in which specific examples are derived from general propositions.

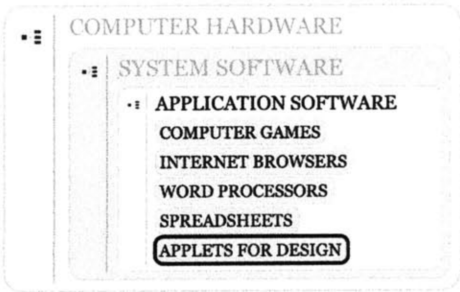


Figure 3-16: Application software is all the computer software that causes a computer to perform useful tasks beyond the running of the computer itself. Applets are small applications that perform only a specific set of tasks.

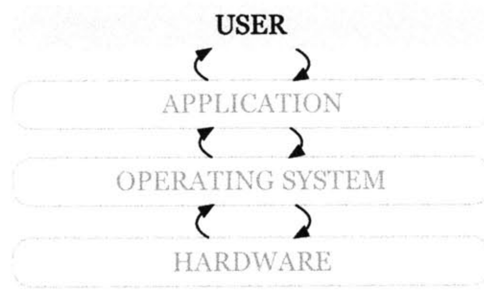


Figure 3-17: The hardware composes the mechanics of the machine, the system software serves the applications, which in turn serve the user. Applet-makers invent new small software applications that serve design.

Application software is all the computer software that causes a computer to perform useful tasks beyond the running of the computer itself. A specific instance of such software is called a software application, application or app. In contrast, system software, manages and integrates a computer's capabilities but does not directly perform tasks that benefit the user. The system software serves the application, which in turn serves the user (Figure 3-17).

Ready-Made Computation (Figure 3-15)

Gesture Interfaces (Level 0), are the most abstract computational engagement with the computer where the user inputs body language. They use gesture recognition algorithms[16] that encode certain human movements and translate them into screen graphics. Graphic Interfaces for design, including CAD packages (Level 1), are also considers passive computation. They provide the user with the ability to input or draw information in the computer screen through traditional physical interfaces – mouse and keyboard or also graphic tablets. Another level down (Level 2), there is the process of automating CAD, that uses high level scripting languages[17] (RhinoScript, RhinoPython, MEL, AutoLISP, etc.) to encode certain repetitive – mostly geometric – routines that humans could otherwise perform one by one in graphical drafting interface. It is considered "end-user programming" (EUD) as it allows people who are not professional developers to modify a software artifact. However, scripting and visual programming within CAD is constrained by the operations available in the geometric kernel that the CAD software operates on, that is why it enables some customized computational structures but it does not fully comply with the rules of self-made computation [Cardoso, 2011].

Self-Made Computation (Figure 3-15)

Starting now from the base (Level 6) there are those practices that develop new physical and philosophical paradigms about new ways to be with the computer and even new structures for its creation. Usually by means of revolutionary publications on ideas like the Quantum Computer[18] first introduced by Manin and

[16] Gesture recognition is a topic in computer science and language technology with the goal of interpreting human gestures via mathematical algorithms. Gestures can originate from any bodily motion or state but commonly originate from the face or hand.

[17] A scripting language or script language is a programming language that supports the writing of scripts, programs written for a software environment that automate the execution of tasks which could alternatively be executed one-by-one by a human operator. Environments that can be automated through scripting include software applications, web pages within a web browser, the shells of operating systems (OS), and several general purpose and domain-specific languages such as those for embedded systems.

[18] The field of quantum computing was first introduced by Yuri Manin in 1980 and Richard Feynman in 1981. A quantum computer

Feynman in the 1980's. A level up (Level 5), we find the exercise of developing – using low level assembly languages – software and hardware according to slight changes to the current computer paradigm. Of course (Level 4), we can choose to accept the computer's hardware as it is and engage in a process of building a customized version of its software – rendering algorithms, 3d geometric kernel, etc. – by direct access to the processor and graphics card of the machine through low level languages (C, C++).

In the core (Level 3) we find applet-making as the process that, I argue, empowers designers to be both in control of the tool they use, and be creative with the computer. It uses high level object-oriented programming languages (such as Java, Processing, C++, Csharp, Ruby, Javascript) that provide minimum functionality in their core, but a vast amount of other libraries based on many different aspects of design and also allow for the creation of new ones. They support the design thinking process of those designers willing to explore design possibilities through computational thinking.

Back to the three modes of computer-aided design, maybe applet-making is the only one where we build a toll to think with and maximize exploration before representing and communicating designs. With traditional CAD, the designer is a tool-user, with scripting a tool-user expert and with applet-making the designer is a tool-maker who is much more in control of the formal descriptions that enable exploration with the computer.

is a computation device that makes direct use of quantum mechanical phenomena, such as superposition and entanglement, to perform operations on data. Quantum computers are different from digital computers based on transistors. Whereas digital computers require data to be encoded into binary digits (bits), quantum computation uses quantum properties to represent data and perform operations on these data.

Chapter 4

Reflections and Contributions

Today, there is the potential to attain far greater control in virtual exploration and production if new man-machine interfaces are devised and engaged. And that this is suddenly possible because of scripting and applet-making cultures. This thesis discusses contemporary questions about self-made computation as a learning and invention process through evidencing a general mindset shift in digital design, from the frustrations of CAD, to the those of scripting. In favor of pioneer understandings like Bill Mitchell's, Paul Cotes's or John Frazer's, but renewing their vision with the improved accessibility to programming through frameworks providing the backbones for the encoding of elements, rules and constraints, that serve open-ended explorations escaping human imagination.

4.1 Contributions

The use of a thesis-wide graphical language to unify, analyze and compare design processes is one of the contributions. It is a graphic guide to help designers think about the use of standard design software, scripting or parametric processes, and self-made computation strategies. It can help future designer-toolmaker identify which are the structures behind staged linear processes coming from engineering problem solving, and those that approach problem-finding in applets or rule-making for design.

The mindset shift addressed here spans education practice and scholarship in the architecture discipline. If architecture is to embark into the world of generative form, its design methods should also incorporate computational processes. While human intuition is the starting point, the computational and combinatorial capabilities of computers can also be integrated [Terzidis, 2007].

Education:
In education there is today an opportunity to teach computational design thinking to support process-based rather than result-oriented design studios. The time we invest today in being expert CAD users, can be accompanied by time learning the tool's abilities and limitations to serve our design intentions and explorations. Maybe that understanding will encourage the doubts we may have about our capability to write code, to describe rules to come to understand phenomena and explore with the computer. In the same way that we used to learn the language of drafting, we can learn now the language that the computer speaks, not as a

substitute of our design thinking but to make the tool a companion, an apprentice instead than a master in human-machine relationships for design.

Practice:

It is important to rise awareness today of the prescriptions and formal structures that operate beneath CAD, and that also constrain scripting. It is a logic tendency for big design software companies to target the most general problems that designers can find while using computers, but for those problems with conflicting constraints, complex requirements and need for exploration, the designer is able to build smaller functionality devices that better serve a non-standard design problem. Even if the professional office calls for shortcuts, the production of tailored tools for specific moments of a project development, can benefit a more controlled and designer-driven process. The building of a growing repertoire of digital design companion modules, can better assist the practice's design intent by accommodating those exploration mechanisms not provided by traditional computer aids.

Scholarship:

Finally, scholarship research on small self-made rule-based modes of dialogue with the computer can lead the way of planting the germ of innovation that the algorithm enables. By acculturating a mode of computational design thinking in research, the mindset shift the designer as a tool-maker will influence education and practice.

4.2 Conclusions

It is important to rise awareness of the contradictions in which designers operate with respect to computer aids to design. Firstly there are critics with strongly negative opinions about scripting who have never themselves scripted [Burry, 2011]. What makes computational logic so problematic for architects is that they have maintained an ethos of artistic sensibility and intuitive playfulness in their practice. In contrast, because of its mechanistic nature, a computational process, e.g. an algorithm, is perceived as a non-human creation and therefore is considered distant and remote [Terzidis, 2007]. There is suspicion and rejection of computation as a design tool that, paradoxically, does not affect traditional CAD – which is no more than geometric solver code wrapped in a UI, and widely used and accepted. In traditional practice, the dominant mode for using computers in design today is a combination of manually driven design decisions and formally responsive computer applications. It raises the problem that neither the designer is aware of the possibilities that computational schemes can produce, nor the software packages are able to predict the moves, idiosyncrasies, or personality of every designer [Terzidis, 2007]. Therefore, the result is a distancing between the potential design explorations and the capacity built into computational tools, missing the opportunity to have a conversation between what designers do best and what computers do best.

However, when comparing contemporary practicing architects we can identify at least one common theme: the use of the computer as an exploratory formal tool and the increasing dependency of their work on computational methods. Through computation, architecture transcends itself beyond the common and predictable. In contrast, through the use of commercial applications and the dependency on their design possibilities, the designers work is at risk of being dictated by the language-tools they use. By unknowingly converting to the constraints of a particular computer applications style, one runs the risk of being associated not with the cutting-edge research, but with a mannerism of hi-tech style [Terzidis, 2007].

Another paradox, that still comes to my mind very frequently, is the contrast between the exactness of programming languages and the woolliness of some design thinking, some say; *how can we design by typing?* In one hand, I do not suggest that we design as we write the applet but, instead, we write code to design with, to see differently, to tailor an environment that suits the design exploration at hand. We build rule-sets with constraints to get a response from the computer that we evaluate and start a back and forth dialogue to explore the design space.
In the other hand – in response to the idea that free-flow creativity can be compromised through the application of the cold hard logic in a script – there is always some logic to find the route to an answer, perhaps yet to be fully identified, using code that foregrounds discovery rather than pre-empts 'the answer' that ought to emerge through its use. If we build our tools in a way that avoids painting the designer into a corner and, instead, provides opportunities for new bifurcations to be encountered along the way, far richer outcomes will be possible for the same investment of time than that which will be arrived at using software merely at face value [Burry, 2011].

There is also the contradiction that the off-the-shelf software that designers use to create with, has been built by computer scientists or engineers and is not originally intended for creative purposes but for representational ones. Some then ask, how do we educate computer scientists to have the requisite design thinking? That is a challenge than I would like to invert. *How can designers have the requisite tool-making thinking to tailor applets for early design?*

Some current problems with scripting that need to be solved before we fully embrace applet-making for design. The originality of scripted designs has not raised at the level of improvement of hardware and software aids. The scripting community, instead of having pluralized from these many innovative and powerful aids, has created around particular approaches such as generative design using genetic algorithms, or agent systems, or some other 'system', usually from the same table. Designers are appropriating scripting systems, not making them [Burry, 2011]. Another issue is that the young scripting generation is more willing to share files on the internet, very different from the instinct of the senior designer to be secretive, that means that even if wheels are not reinvented everywhere as they where in the closeted ateliers of yore, young designers are more inclined to mash up each others' code rather than struggle away from its principles, which could potentially stunt creative growth [Burry, 2011].

However, it is also true that designs aided by programming have been exhibited and published for over 10 years, but is still missing from many architectural education programs. And here I am not talking about technique, but rather referring to the idea that programming skills have noting to do with thinking algorithmically, but if a designer can think in those terms, programming can come more naturally. Traditional design education is based upon a model of investing skills upon an overlay of theoretical and practical application, while the truth is that computational design capacities are only fully unfolded at the convergence of exercising both computational thinking and practice [Menges and Ahlquist, 2012].

4.3 Perspectives

We all seem to be waiting for a natural language and a seamless physical interaction system to appear, one that does not require the designer to prematurely declare priorities in order to comply with structures of computing logic, nor to be forced to interact so palpably with a black box.
–Mark Burry, 2011.

Etymologically, in old latin and greek,"ars"and "tekhne" where bothsynonymsfor craftsmanship. It is today that we separate the concepts by what designers do, and what computers do. What is spontaneous versus what is determined, freedom versus. mechanical operation. *But what if we could use computers as we use craftsman tools?* We are currently not able to define what ideal computer tool we imagine, it should be natural, it should understand our intention, but it still cannot be "made". What is true is that by understanding and customizing current CAD tools, and by proposing computational devices that precede representational processes, we will be less supine to software engineers' computer programs based on a singular view of how designers design.

Will applet-making solve all our computer-aided software frustrations?
No, however, it has been observed here that applet-making, is firstly a viable process by which the designer builds tools for himself to design with, secondly it enables the understanding of how computers work that scales up to comprehending formal descriptions behind bigger software platforms like CAD, and thirdly pushes back representation and communication procedures in design workflows emphasizing early explorations.

Chroniclers of our era may one day ask, "What was computer-aided design? To them, it will just be design [Mitchell and McCullough, 1995]. Through a progressive mindset shift to computational thinking from the idea of designers as software users to designers as also tool-makers. From the computer as a problem-solving calculator, to a companion for problem-finding, we can alter cognitive processes through new technical processes and rethink design today.

Bibliography

R. Aish. *DesignScript: origins, explanation, illustration (White Paper).* Online resource, 2011. Autodesk Research Labs.

R. Alejos and F. Muscinesi. *Hello World! Processing.* Online resource, http://hello-world.cc, 2013. Video Documentary on Creative Coding.

C. Alexander. *Notes on the synthesis of form.* Harvard University Press, 1966.

V. Bazjanac. The promises and disappointments of computer-aided design. In N. Negroponte, editor, *Reflections on computer aids to design and architecture.* Petrochelli/Charter, 1975.

M. Burry. *Scripting Cultures - Architectural Design and Programming.* John Wiley and Sons Ltd., 2011.

M. Burry, S. Datta, and S. Anson. Acadia 2000: Eternity, infinity and virtuality. In *Introductory Computer Programming as a Means for Extending Spatial and Temporal Understanding.* School of Architecture and Building, Deakin University, Australia, 2000.

D. Cardoso. *Architecture and Technology: agency, representation and performance.* Online resource, October 2010. Talk in UFM Guatemala.

D. Cardoso. *Hello (3D) World: An Introduction to Geometry Automation using RhinoScript.* Online resource, 2011. http://mit.edu/ dcardoso/www.

W. Carlson. *A Critical History of Computer Graphics and Animation (Lecture Notes) by Professor Wayne Carlson, PhD, Vice Provost, Dean of Undergraduate Education.* Online resource, 2007. The Ohio State University, Department of Design Technology.

P. Coates. *Programming.Architecture.* Routledge, 2010.

D. Davis, J. Burry, and M. Burry. The flexibility of logic programming: Parametrically regenerating the sagrada familia. *Proceedings of the 16th International Conference on Computer-Aided Architectural Design Research in Asia (CAADIRA)*, pages 29–38, 2011.

W. Forsythe. *Choreographic Objects.* Online resource, 2009. http://synchronousobjects.osu.edu.

J. Frazer. *Accelerating Architecture. Day 4: The Computer's Tale.* Online resource, 2005. Westminster School of Architecture Lecture Series.

N. Gershenfeld. *Fab: The coming revolution on your desktopfrom personal computers to personal fabrication.* Basic Books, New York, 2005.

N. J. Habraken. *Concept Design Games.* National Science Foundation (report), 1987.

G. N. Harper. Bop an approach to building optimization. In *Proceedings of the 1968 23rd ACM national conference*, ACM '68, pages 575–583, New York, NY, USA, 1968. ACM. doi: 10.1145/800186.810621. URL `http://doi.acm.org/10.1145/800186.810621`.

A. Kilian. *Design Exploration through Bidirectional Modeling of Constraints.* PhD thesis, Massachusetts Institute of Technology, 2006.

M. McCullough. *Abstracting Craft. The Practiced Digital Hand.* MIT Press, 1996.

A. Menges and S. Ahlquist. *Computational Design Thinking.* John Wiley and Sons Ltd., 2012.

P. Michalatos. *Topostruct and Graphemes Documentation.* Online resource, 2009. http://www.sawapan.eu.

P. Michalatos and S. Kaijima. Tensor shades. *SIGGRAPH ACM Special Interest Group on Computer Graphics and Interactive Techniques, Art Gallery,* page 15, 2008.

W. J. Mitchell. Vitruvius redux. In Antonsson E.K. and Cagan J., editors, *Formal Engineering Design Synthesis.* Cambridge University Press, 2009.

W. J. Mitchell and M. McCullough. *Digital Design Media.* Van Nostrand Reinhold Company Inc., 1995.

W. J. Mitchell, Liggett R.S., and Kvan T. *The Art of Computer Graphics Programming.* Van Nostrand Reinhold Company Inc., 1987.

L. Mogas-Soldevila. Xecab - xarxa d'espais creatius per a barcelona: Edifici servidor i d'equipaments. Master's thesis, Polytechnic University of Catalonia - Departament de Projectes Arquitectnics, 2009.

C. Mota. The rise of personal fabrication. *Proceedings of the 8th ACM conference on Creativity and cognition,* pages 279–288, 2011.

OpenEndedGroup. *Drawn Together (installation description).* Online resource, 2012. http://openendedgroup.com/artworks/dt.html.

R. Oxman. The new structuralism. *AD Magazine,* 8(4), 2010.

P. Pangaro. *Design for Conversations and Conversations for Design.* Online resource (MIT Design Computation Lecture series), 2011. http://vimeo.com/channels/256598.

S. Papert. *Mindstorms. Children Computers and Powerful Ideas.* Basic Books In., New York, 1980.

A. Pressman. *Designing architecture, the elements of process.* NY Routledge Publisher, 2012.

P. G. Rowe. *Design Thinking.* MIT Press, 1987.

L Santos, J. Lopes, and A.. Leitao. Collaborative digital design: When the architect meets the software engineer. *eCAADe 30, Collaborative Design,* 2:87–96, 2012.

P. Sola-Morales. *Representation in architecture: A data model for computer-aided architectural design.* PhD thesis, Harvard School of Design, 2000.

G. Stiny. What designers do that computers should. In M. McCullough and W. J. Mitchell, editors, *The electronic design studio.* MIT Press, 1990.

G. Stiny. The pedagogical grammar. In A. Tzonis and I. White, editors, *Automation Based Creative Design: Research and Perspectives.* Elsevier Science B.V., 1994.

G. Stiny. *Shape: Talking about Seeing and Doing.* MIT Press, 2006.

G. Stiny. Ice-rays. *Proceeding ACM SIGGRAPH '08, Art and Design Galleries, Design and Computation,* 2008.

G. Stiny. An open conversation about calculation in design. *Dosya Computational Design UCTEA Chamber of Architects Journal,* 1(29):6–11, 2012.

I. Sutherland. Sketchpad: A man-machine graphical communication system. *Proceedings of the AFIPS Spring Joint Computer Conference,* pages 329–346, 1963.

K. Terzidis. *Algorithmic Architecture.* Architectural Press, 2006.

K. Terzidis. Digital design: Ideological gap or paradigm shift? *Proceeding SIGRADI'07 - Theory and Education, Mexico City*, 2007.

P. von Buelow. *Genetically Engineered Architecture: Design Exploration with Evolutionary Computation.* VDM Verlag Dr. Muller, 2007.

J. Weizenbaum. *Computer power and human reason: from judgment to calculation.* W. H. Freeman, San Francisco, 1976.

C. J. Williams. The analytic and numerical definition of the geometry of the british museum great court roof. In M. Burry, S. Datta, A. Dawson, and A.J. Rollo, editors, *Mathematics and Design*. Deakin University, Geelong, Victoria 3217, Australia, 2001.

R. Woodbury, R. Aish, and A. Kilian. Some patterns for parametric modeling. *Proceedings of 27th ACADIA Conference, Association for Computer Aided Design in Architecture*, 2007.